Ereignisorientierte Computersimulation mit ODEMx

Ronald Kluth
Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens

Institut für Informatik

Lehrstuhl für Systemanalyse

Vorwort

Anfang der 90er Jahre entstand am Institut für Informatik (damals noch Fachbereich Informatik) der Humboldt-Universität eine Programm-Bibliothek zur objektorientierten Simulation von Systemen, die die bis dahin erworbenen Erfahrungen und Methoden aus der Nutzung der Sprache *Simula* in der gerade neu aufkommenden Sprache *C++* umsetzte. Diese Bibliothek namens *ODEM (Object Oriented Discrete Event Modelling)* unterstützt sowohl zeitdiskrete als auch zeitkontinuierliche Computersimulationen abstrakter und realer Systeme. Sie erlangte recht bald eine zentrale Position sowohl in den Forschungsprojekten als auch in der Lehre des Lehrstuhls 'Systemanalyse'.

Als erstes größeres C++-Projekt (ca. 16.000 Codezeilen) reflektierte ODEM zugleich die Unzulänglichkeiten der damaligen Gestalt der Sprache C++, der verfügbaren Compiler aber auch unserer Sprachfertigkeiten. Neben graduellen Verbesserungen (bug fixes und Erweiterung der Portierbarkeit auf weitere Prozessorarchitekturen) erfolgte um 2002 eine rigorose Revision der gesamten Implementierung im Rahmen einer Diplomarbeit am Lehrstuhl [Ge03], die sich vor allem dem adäquaten Einsatz der gewachsenen Ausdrucksstärke von C++ (vor allem stabile Mehrfachvererbung und Templates) widmete. Das Resultat dieser Überarbeitung ist seit dieser Zeit unter dem Namen ODEMx (*extended*) auch bei *SourceForge* verfügbar. Die ursprünglich in ODEM realisierte Möglichkeit, Systemzustandsänderungen sowohl prozess- als auch ereignisorientiert nachzubilden, wurde beim Redesign in guter Absicht zugunsten eines durchgängig prozessorientierten Ansatzes aufgegeben.

Bei der Anwendung von ODEMx auf jüngst erschlossene Problembereiche (Modellierung und Simulation verteilter Sensor-Netzwerke zur Erdbeben-Frühwarnung) erwies sich der Verzicht auf leichtgewichtige Ereignisse jedoch als entscheidendes Defizit bei der Skalierung der Anzahl von Modellkomponenten. Im Rahmen der vorliegenden Arbeit wird nun die Bibliothek ODEMx erneut um ein Eventkonzept erweitert, so dass zusätzlich in der Simulation die Ereignisorientierung sowie eine Mischung aus beiden Konzepten durch die Bibliothek unterstützt wird. Auf der Grundlage von Events wird auch das aus der Vorgängerbibliothek ODEM bekannte Timer-Konzept reimplementiert. Die mit dieser Arbeit beschriebene Bibliothekserweiterung wird unter dem Namen *ODEMx v2.0* geführt.

Klaus Ahrens
Joachim Fischer
Ronald Kluth
Berlin, im November 2007

Inhaltsverzeichnis

1 Einleitung

1.1 Thema und Motivation

ODEMx ist eine C++-Klassenbibliothek, die am Lehrstuhl für Systemanalyse zur computergestützten Modellierung und Simulation entwickelt wurde [Ge03]. Das Thema der Arbeit ist die Erweiterung dieser Simulationsbibliothek um ein ereignisorientiertes Modellierungsparadigma.

Bisher lassen sich Systemmodelle in ODEMx ausschließlich auf Grundlage eines prozessorientierten Ansatzes oder Paradigmas spezifizieren, das auf der Beschreibung kooperierender sequenzieller Prozesse bei Konsumption einer fiktiven Modellzeit basiert. Es gibt allerdings in der Modellierung auch einen ereignisorientierten Ansatz, bei dem atomare Ereignisse – sogenannte *Events* – die explizite Grundlage für die Verhaltensmodellierung eines Systems bilden [Ze00].

Abhängig vom untersuchten System kann es sinnvoll sein, verschiedene Modellierungsparadigmen alternativ oder in Kombination zu verwenden, mit denen jeweils das gleiche Untersuchungsziel erreicht werden kann. Da die Simulation komplexer Systeme mit bestimmten Paradigmen unter Umständen eine sehr zeitaufwändige Berechnung nach sich ziehen kann, ist es hilfreich oder notwendig, einen anderen Modellierungsansatz zu wählen.

Dem ODEMx-Anwender soll mit Version 2.0 der Bibliothek die Möglichkeit geboten werden, neben generischen Prozessklassen auch generische Ereignisklassen zu verwenden, welche durch die Bibliothek zur Verfügung gestellt werden. Diese bilden im Rahmen der Arbeit auch die Grundlage zur Reimplementierung eines seinerzeit in der Vorgängerbibliothek ODEM [FA96] umgesetzten Wartekonzepts für Prozesse mit Timer-gesteuertem Aufwachen.

Dabei sollen alle neuen C++-Klassen, welche diese Erweiterungen von ODEMx umsetzen, die Protokollierung und Beobachtung von Simulationsereignissen ebenso konsequent unterstützen wie bereits vorhandene Bibliothekskomponenten.

1.2 Überblick

Zur Klärung der Grundlagen wird im nächsten Abschnitt zunächst eine Erläuterung verschiedener Begriffe der Modellierung und Simulation gegeben, auf denen das zweite Kapitel mit einer Einführung in die diskrete Ereignissimulation aufbaut. Weiterhin werden die dafür benötigten Bestandteile und zwei unterschiedliche Modellierungsparadigmen aufgeführt. Das dritte Kapitel befasst sich mit der Entwicklungshistorie der Simulationsbibliotheken ODEM und ODEMx, mit denen diskrete Ereignissimulation in Kombination mit zeitkontinuierlicher Simulation betrieben werden kann, und gibt einen Einblick in die Funktionalität der Bibliotheken.

Der Hauptteil der Arbeit beschreibt das Konzept der ereignisorientierten Computersimulation und seine Implementierung in ODEMx. Das betrifft sowohl die dafür neu eingeführten Klassen als auch Erweiterungen bestehender Bibliotheksbestandteile und ihre nahtlose Integration mit bibliothekseigenen Konzepten zur Datensammlung. Demonstriert wird die Verwendung des Ereigniskonzepts an einem Beispiel und durch die Reimplementierung der Timer-Funktionalität aus der Vorgängerbibliothek ODEM.

Abschließend werden die Ergebnisse der Arbeit kurz zusammengefasst und ein Ausblick auf mögliche zukünftige Verbesserungen gegeben. Der Anhang enthält ergänzende Quellcode-Auflistungen zu den in der Arbeit aufgeführten Klassen und ihren Funktionen.

1.3 Modellierung und Simulation

Die hier gegebene kurze Einführung in die Grundlagen der Simulation und Modellierung basiert auf [FA96], wo dieses Thema ausführlich behandelt wird.

Modelle und ihre Simulation sind für den Menschen Hilfsmittel, um zielgerichtet mit der Realität umzugehen und unterschiedliche Phänomene der menschlichen Erfahrungswelt abstrakt oder vereinfacht repräsentieren zu können. So werden reale oder fiktive Systeme mit Hilfe von Modellen nachgebildet, um sie hinsichtlich eines Untersuchungsziels zu analysieren.

Ein *Modell* stellt dabei meist eine Vereinfachung dar, die lediglich bestimmte untersuchungsrelevante Aspekte des Originalsystems nachahmt. Als *System* bezeichnet man in diesem Zusammenhang Phänomene, die folgende Eigenschaften besitzen:

- *Systemzweck:* es erfüllt eine bestimmte, erkennbare Funktion,
- *Systemstruktur:* es besteht aus Komponenten, die Wirkungsbeziehungen (*Kopplungen*) untereinander haben und so die Funktionalität festlegen,
- *Systemidentität:* ein System ist unteilbar, das Entfernen von Komponenten erlaubt nicht mehr die Erbringung des ursprünglichen Zwecks und verändert somit die Identität.

Bei der Bestimmung der Systemgrenze zu seiner Umgebung sind genau die Kopplungen zu betrachten, welche für die Erbringung des Systemzwecks wesentlich sind. Durch Abstraktion von bezüglich der Systemfunktionalität unwichtigen Einflüssen kann ein System aus seinem Gesamtzusammenhang herausgelöst und gesondert betrachtet werden, wie Abbildung 1.1 veranschaulicht.

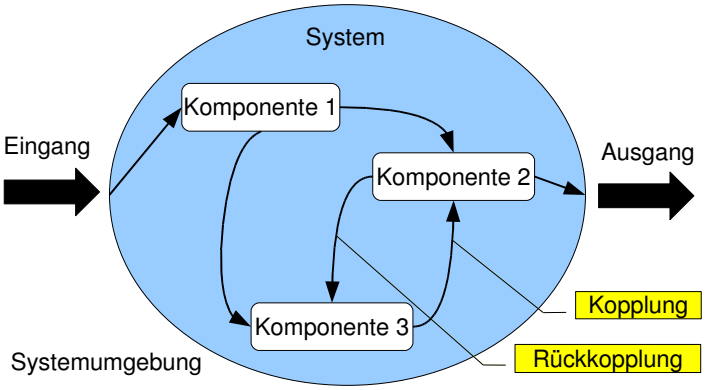

Abbildung 1.1: System mit gekoppelten Komponenten und Umgebung

Verschiedene Bereiche der Wissenschaft untersuchen diverse Arten von Systemen, die beispielsweise physikalische, soziale oder wirtschaftliche Zusammenhänge beschreiben.

In einfachen Systemmodellen können manchmal durch *analytische* Methoden der Mathematik exakte Antworten hinsichtlich eines Untersuchungsziels gefunden werden. Die meisten Systeme sind jedoch zu komplex dafür, weshalb man dann das Hilfsmittel der *Simulation* verwendet. Dieses beschäftigt sich damit, das Verhalten eines Systems im Modell nachzubilden und zu analysieren, um aus den gewonnenen Daten Rückschlüsse auf das Originalsystem zu ziehen, ohne dieses zu beeinflussen.

Computersimulation ist eine Simulationsmethode, die heutzutage mit Hilfe digitaler Computer betrieben wird, um Modelle *numerisch* auszuwerten. Grundlage für die Computersimulation eines Systems ist seine Beschreibung durch formale Methoden, woraus ein

mathematisches Modell resultiert. Der *Simulator* bzw. das Rechenmodell, welches experimentell untersucht wird, ist eine Implementierung des mathematischen Modells unter Nutzung einer geeigneten Beschreibungssprache. Typischerweise betrachtet man bei der Computersimulation *dynamische Systeme*, die in ihrer Struktur und den Eigenschaften ihrer Komponenten veränderlich sind.

Daher spricht man immer von einem *aktuellen Zustand* eines Systems, der sich zu einem Zeitpunkt seiner Struktur entsprechend aus dem Zustand aller seiner zu diesem Zeitpunkt existenten Komponenten ergibt. Dieser Zustand wird mathematisch durch eine minimale Menge voneinander unabhängiger Variablen – sogenannte *Zustandsgrößen* – beschrieben. Die Menge aller möglichen Zustände eines Systems ist sein *Zustandsraum*. Zeitabhängige Systeme werden weiterhin unterschieden in solche, deren Zustand sich in Abhängigkeit der Zeit *kontinuierlich* ändert und andere, in denen nur zu *diskreten* Zeitpunkten Zustandsänderungen auftreten.

Im Allgemeinen unterscheidet sich die vom Computer für die Berechnung einer Simulation benötigte Zeit von der im realen System, weshalb letztendlich zwischen drei Zeitbegriffen unterschieden wird: Realzeit, Modellzeit und Simulationszeit. Die *Realzeit* beschreibt Zeitpunkte, zu denen im realen System Zustandsänderungen auftreten. Zur Nachbildung solcher Zeitpunkte im Modell wird eine Variable verwendet, welche die Realzeit modelliert und deshalb *Modellzeit* genannt wird. Werden nun Zustandsänderungen eines Systems unter Verwendung eines Simulationsmodells auf dem Computer berechnet, so bestimmt die Rechendauer des Computers die benötigte *Simulationszeit*. Ist die benötigte Simulationszeit kleiner oder gleich der Realzeit, spricht man von einer *Echtzeitsimulation*.

2 Diskrete Ereignissimulation

2.1 Einführung

Bei der diskreten Ereignissimulation werden *Aktionen*, welche in Analogie zu realen Phänomenen Änderungen des Zustands und der Struktur des Systemmodells beschreiben, zusammengefasst und ihre Auswirkungen auf das System analysiert. In einem diskreten Ereignismodell finden diese nur zu einzelnen, diskreten Zeitpunkten statt.

Derartige Zustandsänderungen werden üblicherweise als *Ereignisse* bezeichnet. Im Simulator wird die durch ein Ereignis beschriebene Zustandsänderung in Form einer *Ereignisroutine* angegeben, die bei Ereignisrealisierung ausgeführt wird. Grundsätzlich sind dabei zwei Ereignistypen zu unterscheiden. Bei einem *Zeitereignis* ist der Ereigniszeitpunkt bekannt und dient als Auslöser, d.h. sobald die Modellzeit diesen Ereigniszeitpunkt annimmt, wird die durch das Ereignis beschriebene Zustandsänderung im Modell realisiert. Ein *Zustandsereignis* hingegen erfordert die Erfüllung bestimmter Bedingungen, welche vom Systemzustand abhängig sind. Der Ereigniszeitpunkt ist also nicht von vornherein bekannt und gewisse Systemzustände dienen dabei als Auslöser.

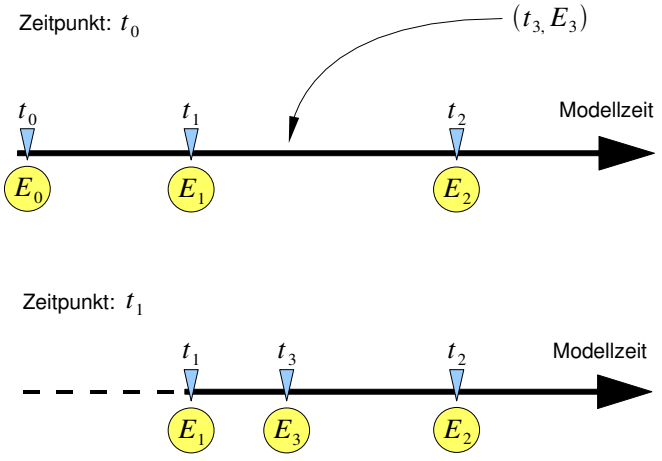

Abbildung 2.1: Zeitfortschritt bei Ereignisrealisierung von E_0

Grundlage für die Berechnung einer ereignisbasierten Computersimulation ist eine sortierte Liste, *Ereigniskalender* genannt, die solche Ereignisse in der Reihenfolge ihres Auftretens enthält. Der Simulator liest diese Liste und führt nacheinander die eingetragenen Ereignisse aus, was Zustandsänderungen im simulierten Systemmodell hervorruft. Im Zuge dessen können weitere neue Ereignisse erzeugt und im Ereigniskalender eingetragen werden. Abbildung 2.1 zeigt die Ausführung des Ereignisses E_0, wodurch ein neues Ereignis E_3 zum Zeitpunkt t_3 eingeplant wird. Die Ausführung eines Ereignisses ist dabei immer zeitlos. Die Modellzeit schreitet also erst dann voran, wenn ein Ereignis abgearbeitet ist. Sie wird jeweils auf die Ausführungszeit des nächsten vermerkten Ereignisses gesetzt, was in Abbildung 2.1 durch einen Sprung von t_0 nach t_1 dargestellt ist. Konfliktsituationen, die dabei durch gleichzeitig auftretende Ereignisse entstehen, lassen sich auf verschiedene Arten lösen:

- Ausführung aller Reihenfolgen durch Aufbau eines Graphen von Folgezuständen,
- Ausführung sequenziell in gleichverteilt zufälliger Reihenfolge,
- Ausführung entsprechend der Reihenfolge, in der die Ereignisse eingetragen wurden,
- Priorisierung von Ereignissen (allerdings besteht weiterhin der zeitliche Konflikt bei gleicher Priorität).

Das erste Verfahren kann zur Entwicklung des gesamten Zustandsraums verwendet werden, was jedoch meist durch das explosionsartige Wachstum möglicher Folgezustände unrealistisch ist, da allein schon der verfügbare Speicher des Simulators Grenzen setzt. So sind für die Praxis nur die verbleibenden drei Ansätze relevant und anwendbar, bei denen während einer Simulation lediglich bestimmte Pfade in der Zustandsrealisierung untersucht werden [FA96].

2.2 Elementare Bestandteile

2.2.1 Simulator

Ereigniskalender Der Simulator verwaltet mindestens eine Liste von Ereignissen. Jedes Ereignis muss einen Ausführungszeitpunkt und eine Ereignisroutine zur Beschreibung der dazugehörigen Aktion besitzen. Der Ereigniskalender wird nach Ausführungszeitpunkten sortiert und Ereignisse werden typischerweise dynamisch mit Fortschritt der Simulation eingetragen.

Uhr Der Simulator muss die aktuelle Modellzeit verwalten, wobei die Zeiteinheiten dem modellierten System entsprechend gewählt werden können. Bei der diskreten Ereignissimulation macht die Zeit im Gegensatz zur Echtzeitsimulation „Sprünge“, da Ereignisse unmittelbar eintreten und zeitlos sind – die Uhr überspringt im Verlauf der Simulation immer die Zeit bis zum Ausführungszeitpunkt des nächsten Ereignisses (siehe Abschnitt 2.1).

Simulationssteuerung Üblicherweise ist ein Scheduler Teil des Simulators. Dieser regelt die Verwaltung der Uhr und des Ereigniskalenders während der Simulation. Er bestimmt das nächste Ereignis und setzt die Uhr auf dessen Ausführungszeitpunkt. Danach wird die Ereignisroutine aufgerufen, welche dann Zustandsänderungen im Systemmodell herbeiführt. Der Scheduler kann auch die Abbruchbedingung überprüfen und Auswertungsmechanismen starten.

Zufallszahlengeneratoren Abhängig vom Systemmodell werden verschiedenste Arten zufälliger Variablen benötigt, um beispielsweise variable Zeitintervalle und Häufigkeiten zu modellieren. Erreicht wird dies durch Verwendung von Pseudozufallszahlen, welche die Simulation vielfältiger Verteilungsarten ermöglichen und dennoch deterministisch reproduzierbar sind, womit Simulationen sich exakt wiederholen lassen.

2.2.2 Simulation

Statistik Die Simulation verwaltet verschiedenste für das Untersuchungsziel relevante statistische Daten über den Simulationsverlauf, die Komponenten des Systems und eingetretene Ereignisse. Dazu gehören meist auch Daten zur Auslastung von Systemkomponenten wie Warteschlangen und Ressourcen, Nutzungsdaten von Zufallszahlengeneratoren und die Bestimmung elementarer Kennwerte wie Mittelwert und Standardabweichung.

Abbruchbedingung Da Ereignisse dynamisch generiert und eingetragen werden, könnte eine diskrete Ereignissimulation theoretisch ewig laufen. Daher muss der Modellierer entscheiden, wann die Simulation endet. Das ist üblicherweise ein Zeitpunkt, ein bestimmter statistischer Wert oder das Erreichen eines festgelegten Systemzustands.

Auswertungsmechanismen Aus den gesammelten Simulationsdaten werden meist durch mehrere Komponenten Protokolle oder Berichte erzeugt, die Informationen zum

Simulationsverlauf zusammenfassen und statistische Daten so aufbereiten, dass der Modellierer im Nachhinein Fragen zum Untersuchungsziel beantworten und Rückschlüsse auf das Systemverhalten ziehen kann [LK00].

2.3 Modellierungsansätze

Es gibt verschiedene Möglichkeiten, diskrete Ereignissimulation durchzuführen, wobei der ereignisorientierte und der prozessorientierte Ansatz zwei Extrema bilden, die an dieser Stelle näher betrachtet werden.

2.3.1 Prozessorientierter Ansatz

Die prozessorientierte Modellierung zeichnet sich dadurch aus, dass Aktionsfolgen bestimmter Systemelemente in Form von Prozessen zusammengefasst werden. Aktionen verbrauchen Modellzeit und während dieser Dauer erfolgen Zustandsänderungen im System. Aktionen von Prozessen sind gekennzeichnet durch Start- und Endzeitpunkt, wobei letzterer dann den Start der nächsten Aktion markiert. So lassen sich Prozesse auch als Folgen von Zeit- und Zustandsereignissen betrachten, zu denen jeweils eine Änderung des Systemzustands eintritt, denn der Abschluss einer Aktion bedeutet den Übergang in einen neuen Zustand.

Es gibt bei diesem Ansatz einen Ereigniskalender, dessen Einträge die Prozessaktivierungszeitpunkte mit Verweisen auf die entsprechenden Prozessobjekte sind, wie in Abbildung 2.2 dargestellt. Der Prozess, welcher aktuell die Steuerung hat, ist solange aktiv, bis er sie freiwillig abgibt – beispielsweise durch Übergang in einen Wartezustand, was dem Modellierer die volle Kontrolle über den Simulationsablauf gibt.

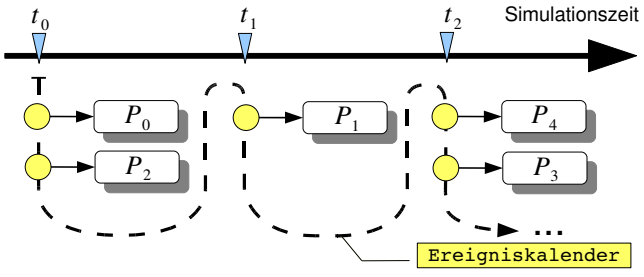

Abbildung 2.2: Ereigniskalender mit Prozessaktivierungszeitpunkten

Prozesse durchlaufen normalerweise verschiedene Phasen in ihrem *Lebenszyklus*, dabei können sie abwechselnd aktiv und passiv sein. In *aktiven Phasen* werden eigene Aktionen ausgeführt, in *passiven Phasen* wird auf das Eintreten externer Ereignisse gewartet, zum Beispiel Zeitereignisse oder die Reaktivierung durch einen anderen Prozess. Warten entspricht bei Prozessen dem Neueintrag im Ereigniskalender zu einem späteren Simulationszeitpunkt, zu dem dann mit der Abarbeitung der Verhaltensbeschreibung fortgefahren wird. Während die Modellzeit bis zu diesem Punkt voranschreitet, können zwischendurch noch weitere Prozesse aktiviert werden und ihre Aktionen ausführen. Durch den Ereigniskalender wird das ganze Verfahren sequenzialisiert.

Mit Prozessen hat der Modellierer stets einen Einblick in partielle Systemzustände und benötigt keinen Gesamtüberblick über das System. Es genügt das Wissen über die Beziehungen zwischen allen Komponenten und jeweils deren Verhaltensbeschreibungen bzw. aktuelle Zustände [FA96].

2.3.2 Ereignisorientierter Ansatz

Dieser Ansatz, auch *next-event simulation* genannt, verfolgt Computersimulation unter Betrachtung und Beschreibung der in einem Systemmodell möglichen Ereignisse. Wie in Abschnitt 2.1 beschrieben, gibt es verschiedene Arten von Ereignissen und ihre Ereignisroutine definiert ihre möglichen Auswirkungen auf den Systemzustand. Jedoch ist an dieser Stelle eine Unterscheidung zwischen Systemverhaltensbeschreibung und der Realisierung eines Systemverhaltens im Verlauf einer Simulation zu machen, denn nicht alle Ereignisse müssen dabei eintreten. Die Realisierung von Ereignissen vollzieht eine Zustandsänderung im Systemmodell und kann nachfolgende Ereignisse beeinflussen.

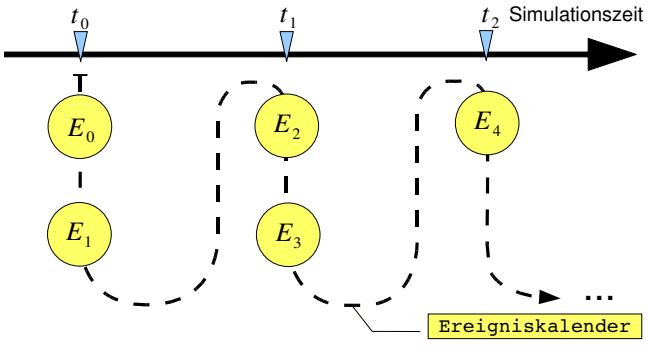

Abbildung 2.3: Ereigniskalender mit eingeplanten Ereignissen

Alle bekannten Ereignisse werden im Ereigniskalender erfasst und in der so gegebenen zeitlichen Abfolge nacheinander abgearbeitet, wie Abbildung 2.3 illustriert. Für einen Schritt bedeutet dies die Entnahme des ersten Eintrags aus dem Ereigniskalender, die Aktualisierung der Modellzeit auf dessen Ereigniszeitpunkt sowie die Ausführung der durch das Ereignis beschriebenen Aktion. Dabei vollzieht sich ein Zustandsübergang im simulierten Modell, es werden neue Ereignisse in der Simulation erzeugt oder vorhandene Einträge verändert, wobei der Simulator Ein- und Ausgaben sowie die Modellbewertung realisiert. Danach wird die Simulation mit dem nächsten Ereignis fortgesetzt.

Beendet wird die Simulation durch ein Zeit- oder Zustandsereignis, welches den Abbruch einleitet. Zur Festlegung von Ereigniszeitpunkten ist Zufälligkeit mit Hilfe von Zufallszahlengeneratoren zu modellieren.

Charakteristisch für diesen Ansatz ist es, eine Abstraktion von einzelnen Systemkomponenten vorzunehmen und dabei alle Zustandsänderungen eines Systems zu betrachten. Allerdings kann diese Sichtweise bei komplexen Systemen schnell zum Verlust der Übersicht führen [FA96].

2.3.3 Zusammenführung

Beide Modellierungsarten unterscheiden sich stark in ihrer Betrachtungsweise eines Systems. Prozessorientierung bietet durch die Betrachtung einzelner Komponenten immer nur den Blick auf einen kleinen Ausschnitt des Systems aus der „Froschperspektive“, macht aber die Modellierung komplexer Systeme gerade dadurch übersichtlicher und leicht erweiterbar. Ereignisorientierung hingegen beschreibt das System aus einer Art „Vogelperspektive“ und verlangt vom Modellierer einen Gesamtüberblick über alle Ereignistypen und Zustandsänderungen bei ihrer Realisierung im System.

Trotz verschiedener Perspektiven sind die beiden Konzepte ineinander überführbar. Eine prozessorientierte Simulation ließe sich in eine ereignisorientierte umwandeln, indem die Aktionsfolgen, die das Verhalten von Prozessen beschreiben, in Ereignisse umgewandelt würden. Im Sinne der Modellierung ist es oft übersichtlicher, Zustandsänderungen als Aktionen in Prozessen zusammenzufassen [FA96].

Die Verwendung von Ereignissen an Stelle von Prozessen ist jedoch in verschiedenen Situationen sinnvoll. Beispielsweise ist es nicht notwendig, Prozesse zu definieren, die nur eine einzelne Aktion modellieren – ein Ereignis, welches das Eintreten der Aktion beschreibt, genügt dafür bereits. Manchmal kann auch das Verhalten einzelner Prozesse unwichtig für das Untersuchungsziel sein und nur die Existenz der Komponente erfasst werden. Dann

kann man den Overhead, der mit Prozessobjekten einhergeht, eventuell durch geschickte Modellierung einsparen. Diese Fragestellung wird auch im Fall der Bibliothek ODEMx zu untersuchen sein (siehe Abschnitt 4.6).

3 ODEM und ODEMx

3.1 Zur Entwicklung der ODEM - Simulationsbibliotheksfamilie

Ein früher Ansatz in der Computersimulation findet sich bereits in den 1960er Jahren, als am Norwegian Computing Centre der Universität Oslo speziell für die diskrete Prozesssimulation die Sprache SIMULA (von *simulation language*) entwickelt wurde, welche allgemein als erste objektorientierte Programmiersprache gilt. Sie beinhaltete insbesondere Konzepte für die Simulation wie einen Ereigniskalender, Scheduling-Funktionen und Prozesse in Form von *Koroutinen*: ein allgemeineres Funktionskonzept, das die Unterbrechung von Abläufen erlaubt und so eine quasi-parallele Ausführung selbstständig agierender Objekte gestattet [Sk97].

Konzeptuell war SIMULA seinerzeit eine überragende Sprache. Das Laufzeitverhalten verfügbarer Programmiersysteme war im Gegensatz dazu allerdings extrem schlecht, was auch Bjarne Stroustrup, der Autor von C++, feststellen musste. Als Konsequenz lag bei seiner späteren Arbeit an C++ der Fokus immer auf Effizenz, weshalb auch Koroutinen nicht Teil des Objekt-Konzepts von C++ sind, sondern separat in einer *task*-Bibliothek implementiert wurden [St97]. Da die ursprüngliche Implementierung der Bibliothek aber einige Nachteile wie fehlende Portabilität und mangelhafte Wiederverwendbarkeit nutzerdefinierter Komponenten aufwies, stellte die weithin portable Reimplementation der *task*-Bibliothek von Hansen eine bessere Lösung zur Umsetzung von Koroutinen in C++ dar [Ha90].

Hansens Version bildete die Grundlage für ODEM, die 1993 am Lehrstuhl für Systemanalyse von Fischer, Ahrens und Witaszek entwickelte C++-Bibliothek zur prozessorientierten Simulation. Die Koroutinen-Implementierung basiert auf der Manipulation des Laufzeit-Stacks und benötigt die Speicherung und Wiederherstellung des aktuellen Prozessorzustands. Durch Martin von Löwis' Portierung des Koroutinenkonzepts für Windows-Plattformen und Sparc-Maschinen [vL92] wurde die Portabilität der Bibliothek verbessert.

ODEM verwendet also Kernkonzepte von Stroustrups *task*-Bibliothek, orientiert sich bei der Nutzerschnittstelle jedoch an der SIMULA-Bibliothek DEMOS [Bi79] mit Konzepten für Prozessobjekte, Warteschlangen, Ressourcen und gepufferte Kommunikation. Die Simulationsbibliothek sollte dem Modellierer das Handwerkszeug für die Prozesssimulation bereitstellen, ohne Kenntnis der zugrundeliegenden Implementierung zu verlangen. Eine Besonderheit von ODEM ist die Möglichkeit, neben zeitdiskreten auch zeitkontinuierliche Prozesse zu simulieren, wobei die kontinuierlichen Zustandsänderungen schrittweise durch fehlersensitive Näherungsverfahren berechnet werden. Auf diese Weise lassen sich zeitdiskrete und zeitkontinuierliche Prozesse gleichzeitig in einer Simulation verwenden [FA96].

Mit der Erweiterung von C++ durch die *Standard Template Library* und mit neuen Erkenntnissen in der Entwicklung von Simulationssystemen offenbarten sich im Laufe der Jahre verschiedene Verbesserungsansätze für ODEM, deren Realisierung 2003 im Rahmen der Diplomarbeit von Ralf Gerstenberger erfolgte [Ge03]. Das Resultat dieser Bemühungen ist die ebenfalls in C++ geschriebene Klassenbibliothek ODEMx, die im Vergleich zu ODEM folgende wesentlichen Neuerungen einführt:

- eine Kapselung von Simulationen und ihren zugeordneten Objekten (Instanzen von C++-Klassen), was im Gegensatz zu ODEM die Schachtelung und Durchführung von mehreren Simulationen in einem Programm gestattet,

- ein dynamisches Kopplungsverfahren von Objekten nach einheitlichen Konventionen, implementiert als C++-Template, das Überwachungsschnittstellen für bestimmte Klassen definiert,

- erweiterte und neue Auswertungsmöglichkeiten für Simulationen durch einen gefilterten Simulationsgesamtüberblick (*Trace*), Berichte für einzelne Modellkomponenten (*Report*) und ein auf der Objektkopplung basierendes Beobachtungsverfahren (*Observation*) für selektive Datenerfassung,

- automatisierte Dokumentationsgenerierung aus Quelldateien, unterstützt durch das Tool *Doxygen*, welches verschiedene Ausgabeformate unterstützt, darunter HTML, LaTeX und PDF, siehe auch [Doxy].

Seit ihrer Entstehung wird die Bibliothek ODEMx am Lehrstuhl für Systemanalyse in der Lehre eingesetzt und hat bereits in verschiedenen praktischen Einsatzgebieten Verwendung gefunden, beispielsweise seit 2006 auch im Projekt *SimRing* [Ev06].

3.2 Struktur und Funktionalität

3.2.1 ODEM

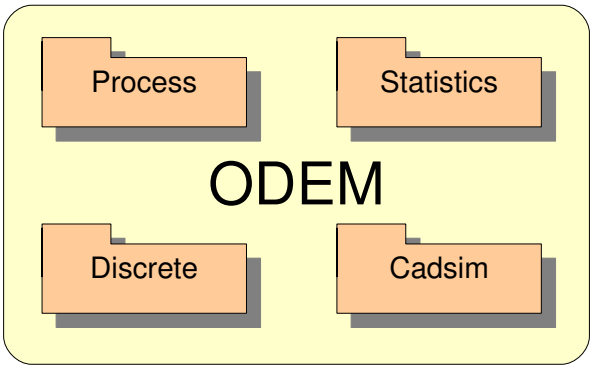

Abbildung 3.1: Aufbau der Bibliothek ODEM

Die Bibliothek ODEM ist modularisiert aufgebaut, wie Abbildung 3.1 verdeutlicht. Intern sind die Bibliotheksklassen in Funktionsbereiche eingeteilt und entsprechend zusammengefasst. Das Modul Process beinhaltet grundlegende Mechanismen wie die Koroutinenimplementierung sowie Klassen zur Prozessverkettung, -synchronisation und -kommunikation. In Statistics sind Generatoren für Pseudozufallszahlen sowie Hilfklassen für die statistische Auswertung implementiert. Die Klassen in Discrete implementieren auf Process aufbauend die zur objektorientierten Prozesssimulation nötigen Konzepte wie Prozesse, verschiedene Warteschlangen, Ressourcen und gepufferte Kommunikation. Dabei werden Auswertungskonzepte für Simulationsereignisse unterstützt. Das Modul Cadsim baut auf Discrete auf und implementiert mit Hilfe eines fehlersensitiven Integrators die Grundlagen zur Verwendung zeitkontinuierlicher Prozesse in ODEM-Simulationen.

3.2.2 ODEMx

Auch die Bibliothek ODEMx ist durch Module strukturiert, dargestellt in Abbildung 3.2. Allerdings wurde eine stärkere Separierung der Klassen vorgenommen. Im Modul Base sind die grundlegenden Bestandteile für die objektorientierte Prozesssimulation gebündelt: Ereigniskalender, Simulationssteuerung, zeitdiskrete und zeitkontinuierliche Prozesse sowie HTML-Ausgabeklassen. Coroutine enthält die Implementierung des Koroutinenkonzepts für quasi-parallele Ausführung von Prozessen und des Koroutinenkontexts, der

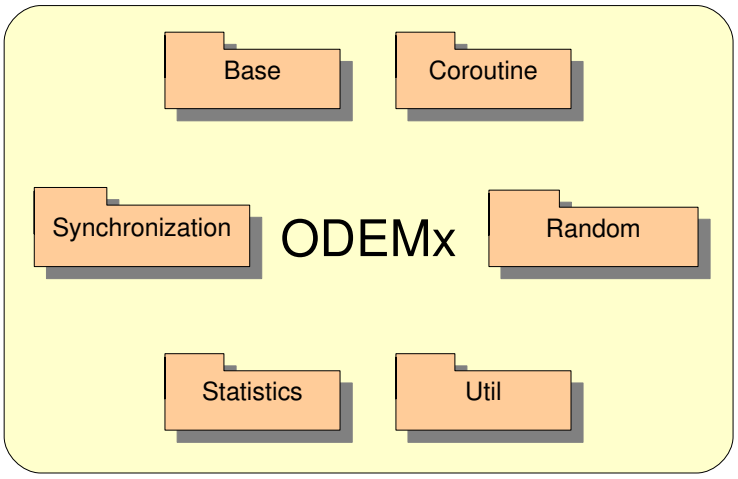

Abbildung 3.2: Modularer Aufbau der Bibliothek ODEMx

die Kapselung von Simulationen ermöglicht. In Synchronization sind verschiedene Warteschlangen, zum Beispiel für eine Master-Slave-Synchronisation oder bedingtes Warten, sowie Klassen zur Modellierung von Ressourcen zusammengefasst. Random stellt Generatoren für Pseudozufallszahlen bereit, womit sich verschiedene ganzzahlige und reelle Wahrscheinlichkeitsverteilungen modellieren lassen. Das Modul Statistics enthält Klassen zur statistischen Berechnung vom einfachen Zähler über Minima, Maxima und gewichtete Mittelwerte bis zur linearen Korrelationsanalyse. Den Abschluss bildet das Modul Util, welches verschiedene Hilfsklassen zur Datensammlung, Ausgabe und Fehlerbehandlung enthält.

3.2.3 Auswertungsmöglichkeiten von ODEMx

In diesem Abschnitt werden die ODEMx-eigenen Techniken der Datensammlung und Datenausgabe gesondert betrachtet, da Erweiterungen der Bibliothek insbesondere auch diese Mechanismen unterstützen müssen. Auf die Umsetzung wird in den Abschnitten 4.5 und 5.5 genauer eingegangen.

Trace Durch das *Trace*-Verfahren werden alle Simulationsereignisse protokolliert. So werden alle Zustandsänderungen eines Systemmodells erfasst, was im Nachhinein die Verhaltensanalyse erleichtert. Dabei spielen Objekte zwei verschiedene Rollen: die des *Trace-Producer*, der Daten sendet, oder die eines *Trace-Consumer*, der Daten empfängt und weiterverarbeitet. Die Unterstützung des Konzepts erfordert von jeder Komponente, dass sie alle wichtigen Informationen durch Funktionsrufe an ein *Trace*-Objekt sendet, welches als Vermittler die Daten an Konsumenten weiterverteilt. Simulationsereignisse werden in

Form von Markierungen versendet, welche jeweils Instanzen der Klasse `MarkType` sind. Bibliotheksseitig wird die Erzeugung einfacher und auch komplexer *Trace*-Markierungen durch viele Funktionen unterstützt.

Observation *Observation* ist ein Konzept, das in Anlehnung an das Komponentenframework *SimBeans* entwickelt wurde. Es basiert auf der Kopplung von Objekten durch Überwachungsschnittstellen, die einheitlichen Konventionen bezüglich Attributsänderungen und Simulationsereignissen folgen. Zwar lassen sich damit auch Interaktionen zwischen Modellelementen modellieren, der hauptsächliche Anwendungsbereich ist jedoch die Erfassung und Auswertung von Simulationsdaten. Auch hierbei gibt es zwei Rollen, die Objekte einnehmen können: die des *Observer*, der ein Objekt überwacht und die des *Observable*, eines Objekts, das überwacht werden kann. Objekte der zweiten Gruppe, deren Funktionen beobachtbar sein sollen, definieren dafür in Form einer `Observer`-Klasse die Schnittstelle, welche zur Datenerfassung verwendet werden kann. Sie müssen zudem vom C++-Template `Observable<>` abgeleitet sein, wobei als Parameter ihre Schnittstellendefinition angegeben wird. Beispielsweise würde eine Klasse `Process` die Beobachter-Schnittstelle `ProcessObserver` definieren und müsste zudem von `Observable<ProcessObserver>` abgeleitet sein.

Report Mit Hilfe des *Report*-Verfahrens bietet ODEMx ein Mittel, um zusammenfassende Berichte über Simulationskomponenten zu erstellen, was vor allem bei Warteschlangen, Ressourcen und Zufallszahlengeneratoren Anwendung findet. Einem *Report*-Objekt werden dabei beliebig viele Reportproduzenten zugeteilt, die auf Anforderung ihre bisher erfassten Daten in Form einer Tabelle an das *Report*-Objekt weitergeben. Die gesammelten Daten betreffen hauptsächlich statistische Werte wie Nutzunghäufigkeit, Mittelwerte sowie Minima und Maxima. Eine umfassende Beschreibung der drei Konzepte findet sich in [Ge03].

4 Implementierung von Ereignissen in ODEMx

Den Schwerpunkt dieser Arbeit bildet die Implementierung einer Basisklasse für die ereignisorientierte Modellierung sowie die Anpassung vorhandener Bibliothekselemente an das neue Konzept. Neue Klassen und Funktionen müssen sich nahtlos in die bibliothekseigenen Methoden der Informationsverwaltung einfügen und ihre Protokollierungsmechanismen unterstützen.

4.1 Das Ereigniskonzept in ODEM

Eine Betrachtung der internen Struktur der Bibliothek ODEM zeigt, dass im Ereigniskalender `Sched`-Objekte verwaltet werden. Dadurch können also unter Verwendung polymorpher Zeiger neben `Process`-Ableitungen auch Objekte anderer Klassen im Ereigniskalender eingeplant werden. Dies ist in ODEM der Fall bei `Timer`-Objekten (siehe Abbildung 4.1).

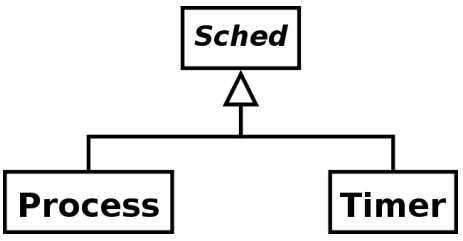

Abbildung 4.1: Ausschnitt der Klassenhierarchie von ODEM

Ableitungen der Klasse `Process` bilden in ODEM den Kern der Modellierung. Wie in Abschnitt 2.3.1 beschrieben, gibt es in ihren Lebenszyklen aktive und passive Phasen, wobei Wartephasen neben Zustandsereignissen auch durch Zeitereignisse unterbrochen werden können, was in ODEM mit der Klasse `Timer` realisiert ist, deren Objekte nur eine atomare

Aktion, also ein Ereignis, verwirklichen. Somit gab es an dieser Stelle bereits in ODEM eine konzeptuelle Mischung aus prozessorientierter und ereignisorientierter Simulation, wobei Zeitereignisse jedoch nur zur Laufzeitverbesserung eingeführt wurden [FA96].

4.2 Ansatz für Ereignisse in ODEMx

Die oben beschriebene Idee ist der Ausgangspunkt für die Implementierung allgemeiner Ereignisse in ODEMx. Weil sich die interne Struktur von ODEMx in diesem Punkt deutlich von der Organisation in ODEM unterscheidet, sind eine Reihe von Anpassungen vorzunehmen, um neben Prozessobjekten auch Ereignisobjekte in der Simulation zu verwenden. Ereignisse stellen ein grundlegendes Modellierungskonzept dar, weshalb alle in diesem Kapitel beschriebenen Modifikationen an ODEMx nur das Modul Base betreffen, wo das Konzept eingeordnet wird (siehe Abschnitt 3.2.2).

Da der Ereigniskalender bisher nur Zeiger des Typs `Process` verwalten kann, ist es notwendig, eine gemeinsame Basisklasse für Prozesse und Ereignisse zu definieren. Zur Weiterverwendung des bisherigen Codes bietet es sich an, eine solche Basis in Form eines Interfaces bereitzustellen. In C++ ist das problemlos durch eine abstrakte Klasse umzusetzen, von der dann die Prozess- und Ereignisklassen abgleitet werden können. Wegen der in C++ erlaubten Mehrfachvererbung kann die restliche Klassenhierarchie, in der Prozesse eingebettet sind, unangestastet bleiben, wie Abbildung 4.2 zeigt.

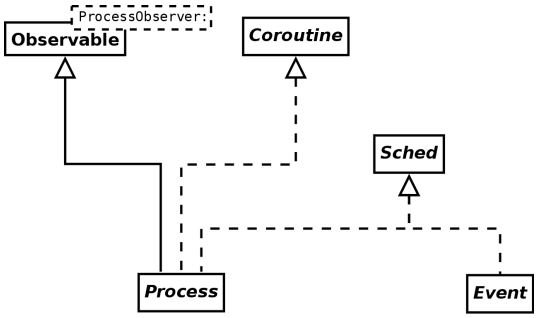

Abbildung 4.2: Erweiterung der Klassenhierarchie von ODEMx

Zur Ereignisimplementierung wird die Interface-Klasse `Sched` eingeführt, von der dann neben `Process` auch die Basisklasse `Event` für nutzerdefinierte Ereignisse abgeleitet wird. Nach einer Anpassung des Ereigniskalenders kann dieser dann durch Verwendung polymorpher `Sched`-Zeiger alle derartigen Objekte in einem gemeinsamen Ereigniskalender verwalten. Insbesondere an der Klasse `Simulation`, welche die Funktionalität des Ereigniskalenders erbt und die Simulation berechnet, sind mehrere Änderungen notwendig.

4.3 Neue und veränderte Klassen

4.3.1 Das Interface `Sched`

```
class Sched: public virtual TraceProducer {
public:
    enum SchedType { PROCESS, EVENT };
public:
    Sched( SchedType st ): schedType( st ) {}
    virtual ~Sched() {}
    virtual SimTime getExecutionTime() const = 0;
    virtual SimTime setExecutionTime( SimTime time ) = 0;
    virtual Priority getPriority () const = 0;
    virtual Priority setPriority ( Priority newPriority ) = 0;
    virtual void execute() = 0;
    SchedType getSchedType() const;
private:
    SchedType schedType;
    friend bool operator <( const Sched& first, const Sched& second );
};
```

Quelle 4.1: Interface `Sched`

Als Basisklasse für Scheduling-Objekte wird auch in ODEMx die abstrakte Klasse `Sched` eingeführt, die das Interface in Form von rein virtuellen Funktionen vorgibt. In C++ dürfen und können von solchen Klassen keine Objekte erzeugt werden. Sollen Ableitungen abstrakter Klassen instanzierbar sein, so müssen darin alle rein virtuellen Funktionen implementiert werden, wodurch eine Vorgabe für die Funktionalität der erbenden Klassen gemacht werden kann.

Das in Quelle 4.1 dargestellte Interface `Sched` ist abgeleitet von `TraceProducer`, um sicherzustellen, dass jede Ableitung über `getTrace()` den Zugang zu einem `Trace`-Objekt implementiert und eine Bezeichnung hat, die durch `getLabel()` abrufbar ist. Deklariert werden weiterhin jene Funktionen, die in der ODEMx-Klasse `ExecutionList` benötigt werden, da diese den Ereigniskalender implementiert und den Zugriff darauf regelt. Das sind `set/getPriority()` und `set/getExecutionTime()`. Daneben bedarf es zusätzlich der Definition des Vergleichsoperators `operator <()`, um `Sched`-Objekte nach Ausführungszeit und Priorität sortieren zu können. Die Funktion `getSchedType()` wird bei der Simulationsberechnung abgefragt, weil Prozesse und Ereignisse in der Ausführung unterschiedlich behandelt werden, auch wenn sie polymorph über den gleichen Funktionsruf `execute()` zu starten sind. Der Typ wird im Konstruktor gesetzt und kann entweder

PROCESS oder EVENT sein, entsprechend der Definition in enum SchedType.

4.3.2 Die Klasse Event

```
class Event: public Sched, public virtual DefLabeledObject,
             public Observable<EventObserver> {
public:
    Event( Simulation* sim, Label l, EventObserver* eo = 0 );
    virtual ~Event();
    virtual Priority getPriority () const;
    virtual Priority setPriority ( Priority newPriority );
    SimTime getExecutionTime() const;
    void schedule ();
    void scheduleIn( SimTime t );
    void scheduleAt( SimTime t );
    void scheduleAppend();
    void scheduleAppendIn(SimTime t);
    void scheduleAppendAt(SimTime t);
    void removeFromSchedule();
    bool isScheduled ();
     ...
protected:
    friend class ExecutionList ;
    friend class Simulation ;
    SimTime setExecutionTime( SimTime time );
    virtual void eventAction() = 0;
    void execute ();
private:
     ...
};
```

Quelle 4.2: Klassendeklaration Event

Genau wie Prozesse müssen Ereignisobjekte die für den Schedulingmechanismus notwendigen Eigenschaften haben, damit sie im Ereigniskalender einsortiert werden können. Um dem Rechnung zu tragen, erbt und implementiert die in Quelle 4.2 dargestellte Klasse Event das Interface Sched. Auch bei Ereignissen ist das potenzielle Verhalten in nutzerdefinierten Ableitungen festzulegen, nämlich durch die vorgegebene rein virtuelle Funktion eventAction(), die in jeder Event-Ableitung durch den Modellierer zu definieren ist. Die aufgelisteten Funktionen schedule... und scheduleAppend... ermöglichen das Scheduling von Ereignissen nach dem *last-in-first-out (LIFO)*- bzw. *first-in-first-out (FIFO)*-Prinzip. Ereignisobjekte haben lediglich zwei Zustände; sie sind entweder eingeplant

oder nicht, was durch `isScheduled()` abzufragen ist.

4.3.3 Die Klasse `ExecutionList`

```
class ExecutionList : public Observable<ExecutionListObserver> {
public:
    void addSched(Sched* p);
    void insertSched(Sched* p);
    void insertSchedAfter(Sched* p, Sched* previous);
    void insertSchedBefore(Sched* p, Sched* next);
    void removeSched(Sched* p);
    Sched* getNextSched();
    void printExecutionList ();
    ...
private:
    std :: list <Sched*> l;
    friend class Event;
    ...
};
```

Quelle 4.3: Änderungen an `ExecutionList`

Die Klasse `ExecutionList` implementiert den Ereigniskalender in ODEMx. Sie wurde dahingehend abgeändert, dass die interne Ereignisliste mit `Sched`-Zeigern arbeitet. Die ursprünglich auf Prozesse bezogenen Memberfunktionen wurden umbenannt, wie Quelle 4.3 zeigt, um ihre geänderte Funktionalität auch im Namen zu reflektieren.

Da verschiedene Funktionen der Klasse `Event` Zugriff auf private Funktionen des Ereigniskalenders benötigen, ist auch `Event` für `ExecutionList` als `friend` deklariert. Das kann zum Beispiel nötig sein, wenn sich die Priorität eines eingetragenen Objekts ändert. Dann muss zumindest überprüft werden, ob die aktuell gegebene Ausführungsreihenfolge angesichts der veränderten Priorisierung noch korrekt ist, oder ob neu sortiert werden muss.

4.3.4 Die Klasse `Simulation`

`Simulation` ist die ODEMx-Klasse, die sich mit dem Großteil der Organisation einer Simulation befasst. Sie bildet den Kontext für alle Modellelemente und initialisiert alle für den Simulationsbeginn nötigen Objekte. Weiterhin übernimmt diese Klasse auch die Funktion des Schedulers. So sortiert und liest die `Simulation` den Ereigniskalender,

```
class Simulation : public ExecutionList, public CoroutineContext, public DistContext,
                   public virtual LabelScope, public Trace, public virtual TraceProducer,
                   public virtual StatisticManager, public Observable<SimulationObserver> {
public:
    Sched* getCurrentSched();
    ...
private:
    friend class Event;
    void setCurrentSched( Sched* s );
    void compSimulation(bool inSimulation = true);
    void exec();
    bool executingEvent;
    Sched* currentSched;
    ...
};
```

Quelle 4.4: Änderungen an `Simulation`

den sie durch Ableitung von `ExecutionList` inne hat, und führt eingetragene Objekte dem `SchedType` entsprechend aus, wobei sie die Simulationszeit aktualisiert. Nur wenn als nächstes ein Prozess auszuführen ist, wird durch die Aktivierung der Koroutine ein Stackwechsel eingeleitet – Ereignisaktionen werden auf dem aktuellen Laufzeit-Stack ausgeführt. Quelle 4.4 stellt die Änderungen an der Klasse `Simulation` dar. Das aktuelle Objekt, sei es Prozess oder Ereignis, wird im Memberdatum `currentSched` gespeichert, worauf über `set/getCurrentSched()` zugegriffen werden kann. Zur Erweiterung des Scheduling-Mechanismus für Ereignisse wurde das Memberdatum `executingEvent` hinzugefügt (siehe Abschnitt 4.4.2). Weiterhin wurde `Event` wie `Process` für den Zugriff auf private Funktionen als `friend` deklariert.

4.4 Scheduling und Simulationsberechnung

In diesem Abschnitt wird das Scheduling-Verfahren von ODEMx dargelegt. Dabei wird zunächst genauer auf die bisherige Prozess-Abarbeitung eingegangen, um auf einige besondere Details hinzuweisen. Im Anschluss wird dann die Erweiterung des Verfahrens um Ereignisse erläutert.

Grundsätzlich ermöglicht ODEMx drei unterschiedliche Simulationsmodi: schrittweise Ausführung, zeitbegrenzte Ausführung oder uneingeschränkte Ausführung bis der Ereigniskalender leer ist. Der Ablauf einer Simulation beginnt im C++-Hauptprogramm mit

dem Aufruf einer der Funktionen `step()`[1], `runUntil()` oder `run()`, die ihrerseits `init()` für die Simulation aufrufen und dann die Berechnung mit der Funktion `compSimulation()` starten. In dieser Funktion wird die entsprechende Fortsetzung anhand der verschiedenen Simulationsmodi `STEP`, `UNTILTIME` und `UNLIMITED` entschieden.

4.4.1 Prozesse in ODEMx: Scheduling und Ausführung

Die Ausführung der Objekte aus dem Ereigniskalender erfolgt durch die Funktion `exec()`, welche die Modellzeit aktualisiert und den Prozess durch `Process::execute()` als Koroutine ausführt. Die *zentrale Funktion* ist dabei `Coroutine::switchTo()`, die eine Koroutine initialisiert, den aktuellen Stack – des Hauptprogramms oder Vorgängerprozesses – mittels `Coroutine::saveEnvironment()` sichert und den Lebenszyklus des Prozesses durch `Coroutine::start()` anstößt, denn in dieser Funktion wird `Process::main()` gerufen. Sollte es nicht der erste Aufruf der Koroutine sein, so befindet sich der Prozess bereits in der Abarbeitung seines Lebenszyklus' und der zuvor gesicherte Zustand wird per `Coroutine::restoreEnvironment()` wieder hergestellt.

Das Speichern des Stacks betrifft dabei immer den kompletten Stackbereich zwischen dem `Coroutine::switchTo()`-Aufruf, wo die Koroutine aktiviert wurde, bis zum nächsten Ruf der Funktion `Coroutine::saveEnvironment()`, in der die Koroutine nach Wiederherstellung auch fortgeführt wird – gefolgt vom Stackabbau durch Beendigung von Funktionen, bis wieder `Process::main()` erreicht wird, wo der Lebenszyklus fortgesetzt werden kann.

In ODEMx gibt es keinen Scheduler, der in einer Schleife den Ereigniskalender abarbeitet. Stattdessen geben immer Aufrufe von `compSimulation()` den Anstoß für die Fortsetzung des Simulationsablaufs. In allen Situationen, wo Prozesse unterbrochen oder beendet werden, folgt ein Aufruf von `compSimulation()`. Das ist der Fall bei allen Prozess-Scheduling-Funktionen und auch am Ende des Lebenszyklus', nachdem `Process::main()` abgearbeitet worden ist. Dadurch ergibt sich ein Zyklus von Funktionsaufrufen, der in ODEMx den sonst üblichen Scheduler ersetzt.

4.4.2 Ereignisse in ODEMx: Scheduling und Ausführung

Ableitungen der Klasse `Event` modellieren atomare Aktionen, die durch die Memberfunktion `Event::eventAction()` implementiert werden. `Event`-Objekte können auf dem

[1] Memberfunktionen der Klasse `Simulation` sind in diesem Abschnitt ohne Geltungsbereich `Simulation::` aufgeführt

Stack des vorher aktiven Prozesses ausgeführt werden, weil lediglich der Start oder die Fortführung einer Koroutine die Auslagerung des alten Programmstacks und Einlagerung des Stacks des neuen Prozesses verlangt. Die von der `Simulation` gerufene Funktion `Event::execute()` ist folgendermaßen implementiert:

```
void Event::execute() {
    if ( !scheduled )
        fatalError ("Event::executeEvent(); event is not in schedule", -1);
    env -> removeSched( this );
    scheduled = false;
    // trace and observer ...
    env -> setCurrentSched( this );
    eventAction();
}
```

Das `Event`-Objekt wird dadurch aus dem Ereigniskalender entfernt und als aktuelles Objekt bei der Simulation registriert. Danach wird die für das Ereignis definierte Aktion durch Ruf von `Event::eventAction()` ausgeführt.

Der naheliegendste Ansatz, die Ausführung von `Event`-Objekten in den Simulationsablauf einzubinden, wäre eine an Prozessen orientierte Umsetzung. Eine Unterscheidung nach dem `SchedType` wäre nur nötig, um die Protokollierung typbezogen vorzunehmen. Die Ausführung eines `Event` würde dabei auch in `Simulation::exec()` unter Ausnutzung der Polymorphie des `Sched`-Zeigers `next` erfolgen und könnte so aussehen:

```
void Simulation::exec() {
    ...
    Sched* next = ExecutionList::getNextSched();
    if ( next -> getSchedType() == Sched::PROCESS ) {
        // trace and observer for process ...
    }
    else { // Sched::EVENT
        // trace and observer for event ...
    }
    next -> execute();
}
```

Da es in ODEMx keinen separaten Scheduler gibt, verlangt die Berechnung der Simulation für Ereignisse eine Lösung unter Verwendung von `compSimulation()`, wenn man die Grundstruktur der Bibliothek beibehalten will. Den gleichen Ansatz wie Prozesse zu verwenden, in dem alle Scheduling-Funktionen am Ende `compSimulation()` aufrufen, führt allerdings nur kurzfristig zum Erfolg. Zunächst funktioniert der Kreis zwischen `compSimulation()`, `exec()`, `Event::execute()`, `Event::eventAction()`,

`Event::schedule...` und `compSimulation()` zwar insofern korrekt, dass in der Simulationsberechnung die Ereignisse und Prozesse in der erwarteten Reihenfolge abgeareitet werden.

Eine Untersuchung zeigt aber, dass sich in einigen Fällen der Laufzeit-Stack des Simulationsprogramms immer weiter aufbaut, ohne jemals abgeräumt zu werden. Das passiert vor allem dann, wenn Ereignisse häufig wieder sich selbst oder andere Ereignisse im Ereigniskalender einplanen. Prozesse verursachen dieses Problem nur deshalb nicht, weil bei jeder Prozessausführung `Coroutine::switchTo()` gerufen wird. Es findet an dieser Stelle dann entweder eine Koroutinenaktivierung mit Stackwechsel statt oder der Laufzeit-Stack des aktuellen Prozesses wird durch Rückkehr aus den gestarteten Funktionen wieder rückwärts abgerollt, bis `Process::main()` weiterläuft.

Es ist somit klar, dass `Event`-Objekte eine andere Behandlung benötigen. Mit dem Wissen, dass Prozesse bei Passivierung immer wieder `compSimulation()` rufen, kann man die eventuell zwischen Prozessen eingeplanten Ereignisse an dieser Stelle gesondert behandeln. Zur Umsetzung wird `compSimulation()` um eine Schleife erweitert, in der alle vor dem nächsten Prozess eingetragenen Ereignisse nacheinander auf dem selben Stack ausgeführt werden. Verkompliziert wird die an sich einfache Schleife allein durch die Berücksichtigung der verschiedenen Simulationsmodi:

```
void Simulation::compSimulation( bool inSimulation /*= true*/ ) {
    if ( executingEvent ) return;    // ensure completion of eventAction()

    // execute events scheduled before next process
    while( !ExecutionList::isEmpty() &&
           getNextSched() -> getSchedType() == Sched::EVENT ) {
        switch( simMode ) {
        case CONTINUOUS:
            if ( !isStopped ) exec();
            break;
        case STEPPING:
            if ( !isStopped ) {
                if ( inSimulation )
                    switchTo(); // called from process scheduling fct, jump to main()
                                // remember this position for the current process
                else exec();    // coming from main(), execute Event
                return;         // do nothing else after returning
            }
            break;
        case UNTILTIME:
            if ( !isStopped && getTime() <= endTime ) {
                if ( getNextSched() -> getExecutionTime() > endTime ) {
```

```
                    if ( inSimulation ) switchTo(); // called from Process scheduling
                    else return;                    // return to main()
                }
                else exec();                        // executionTime <= endTime
            }
        }
    }  // done with events
    // proceed with processes
    ...
}
```

Da `compSimulation()` bei Simulationsstart und bei jeder Prozesspassivierung bzw. Prozessterminierung gerufen wird, ist sichergestellt, dass alle zwischen Prozesswechseln eingeplanten Ereignisse auch abgearbeitet werden. Ein Aufruf von `compSimulation()` ist so in Memberfunktionen von `Event` nicht notwendig. Dadurch muss jedoch noch eine weitere Situation betrachtet werden. Ereignisse sollen in `Event::eventAction()` auch Prozesse im Ereigniskalender einplanen können. Der Ruf einer Prozess-Scheduling-Funktion führt jedoch erneut zur Ausführung von `compSimulation()`, wodurch die gestartete `eventAction()` nicht sofort beendet wird, sondern erst beim Abbau des Laufzeit-Stacks. Es muss also gesichert werden, dass die Ausführung von Ereignissen tatsächlich atomar ist. Dies kann in `Simulation` unter Verwendung des Memberdatums `executingEvent` geschehen und erfordert eine weitere Modifikation bei der Ereignisausführung in `exec()`, wonach die Funktion dann etwas von der eingangs angestrebten Variante abweicht:

```
void Simulation::exec() {
    Sched* next = ExecutionList::getNextSched();
    if ( next -> getExecutionTime() < now ) {
        fatalError ( "exec(); ExecutionList corrupted: ending simulation", -1 );
    }
    now = next -> getExecutionTime();
    switch ( next -> getSchedType() ) {
    case Sched::PROCESS:
        // trace and observer for process ...
        next -> execute();
        break;
    case Sched::EVENT:
        // trace and observer for event ...
        executingEvent = true;
        next -> execute();
        executingEvent = false;
        break;
    }
}
```

In `compSimulation()` wird zu allererst anhand von `executingEvent` festgestellt, ob gerade ein `Event` aktiv ist, in welchem Fall die Funktion nicht ausgeführt wird, sondern sich beendet. Damit schreitet die Simulationsberechnung erst dann voran, wenn die Ereignisaktion komplett abgearbeitet worden ist.

4.5 Unterstützung der Auswertungsmöglichkeiten

Es ist wichtig, dass bei Simulationen möglichst viele Daten gesammelt werden, um das anhand des Modells simulierte Verhalten genau analysieren zu können. Die in Abschnitt 3.2.3 beschriebenen Auswertungsmechanismen müssen also auch durch neue Komponenten ausgiebig genutzt werden. Die Quellcode-Auflistungen der im Folgenden beschriebenen Beobachter-Schnittstellen und Trace-Markierungen sind in Anhang A.1 zu finden.

Simulation Die Klasse `Simulation` bildet den Kontext für alle Komponenten eines Systemmodells. Sie erbt die Funktionalität von `Trace` und ist somit der Empfänger aller während des Simulationslaufs versendeten *Trace*-Markierungen. Gleichzeitig unterstützt sie sowohl die Funktionalität eines `TraceProducer` als auch eines `Observable<>`, womit auch die Funktionsaufrufe dieser Klasse nachvollziehbar protokolliert werden. Es ergibt sich bei der Schnittstelle `SimulationObserver` nur eine einzige Erweiterung bezüglich `Event`, über die sich die Auslösung von Ereignissen in der Simulation beobachten lässt. Ebenso wird bei der Ausführung von `Event::executeEvent()` eine zusätzliche *Trace*-Markierung versandt. Da bei vielen komplexen *Trace*-Markierungen auch Partnerprozesse aufgeführt sind, wurden zur Erweiterung des *Trace*-Mechanismus für Ereignisse die Funktionen `setCurrentSched()` und `getCurrentSched()` hinzugefügt, um auch Ereignisse als Aufrufpartner protokollieren zu können.

ExecutionList Die Klasse des Ereigniskalenders sendet keine *Trace*-Markierungen. Sie besitzt jedoch eine `ExecutionListObserver`-Schnittstelle, welche an die abgeänderten Funktionen angepasst wurde, die aus der Umstellung auf `Sched`-Objekte resultieren.

Sched Da die relevanten Memberfunktionen der Klasse `Sched` rein virtuell und nicht implementiert sind, werden weder *Observation*-Ereignisse noch *Trace*-Markierungen gesendet. Bei derartigen Interfaces ist es sinnvoll, die Daten in ihren Ableitungen zu erzeugen, wo der Klassentyp feststeht und die vorgegebenen Funktionen implementiert sind. In ODEMx wird dieser Weg beschritten, wie der nächste Absatz zu `Event` und auch

Abschnitt 5.5 beschreiben. Durch Erben der Basisklasse `TraceProducer` wird die rein virtuelle Funktion `getTrace()` vereinbart, die von Ableitungen zu implementieren ist.

Event Als Ableitung von `Sched` implementiert die Basisklasse für Ereignisse sowohl die Funktionalität eines `TraceProducer` als auch die eines `Observable<>`. Dafür wurde die Schnittstelle des `EventObserver` definiert. Alle simulationsrelevanten Memberfunktionen senden sowohl *Trace*-Markierungen als auch *Observation*-Ereignisse während des Simulationslaufs. Das umfasst die Erzeugung und Vernichtung von `Event`-Objekten, alle Scheduling-Funktionen, die Ereignis-Ausführung sowie Parameteränderungen bei Priorität und Ausführungszeit.

4.6 Vergleichendes Anwendungsbeispiel

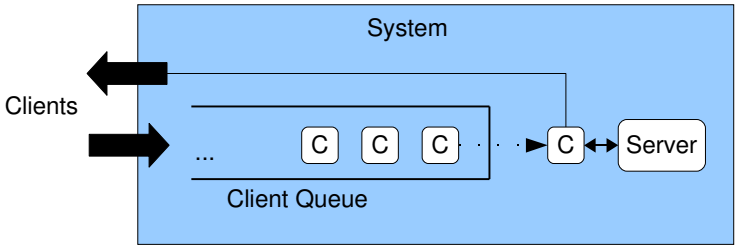

Abbildung 4.3: Schematische Darstellung eines Client-Server-Systems

Das in Abbildung 4.3 dargestellte einfache *Client-Server*-System umfasst als Komponenten einen *Server* und beliebig viele zufällig anfragende *Clients*. Das Untersuchungsziel sei in diesem Zusammenhang lediglich die Frage, wie lange einzelne Clients durchschnittlich auf Bedienung warten müssen.

Es ergibt sich folgendes Systemverhalten: anfragende Clients überprüfen, ob der Server verfügbar ist und reihen sich ansonsten in eine Warteschlange ein. Der Server arbeitet sequentiell die eintreffenden Client-Anfragen ab und wechselt dabei zwischen den Zuständen *beschäftigt* und *frei*, wobei die genauen Aktionen des Servers nicht interessieren, sondern eine gleichverteilte zufällige Abarbeitungszeit verwendet wird.

4.6.1 Prozessbasiertes Modell

Um das Zusammenspiel der Komponenten `Server` und `Client` zu modellieren, wird die Bibliothekskomponente `Waitq` verwendet, die eine *Master-Slave*-Synchronisation und au-

tomatische Report-Erzeugung unterstützt. Der `Server` reiht sich immer wieder durch `coopt()` als *Master* ein und wartet auf Synchronisation mit einem `Client`. Sobald die Zuteilung erfolgt ist, wird durch `holdFor()` die Arbeitsdauer simuliert, wonach der aktuelle `Client` aktiviert wird und das System verlässt.

```
class Server: public Process {
public:
    Server( Simulation* sim, Label l, Rdist* rand ):
        Process( sim, l ), servicePeriod( rand ) {}
    virtual int main() {
        Process* currentClient;
        while( true ) {
            currentClient = ::clientQ -> coopt();
            holdFor( servicePeriod -> sample() );
            currentClient -> activate();
        }
        return 0;
    }
private:
    Rdist* servicePeriod;
};
```

Quelle 4.5: `Server`-Prozess

Ein `Client` hingegen reiht sich als *Slave* in die `Waitq` ein und wartet auf Aktivierung durch den `Server`. Nach Aktivierung verlässt der `Client` das System, was durch sofortige Beendigung seines Lebenszyklus' modelliert ist. Einzig die Wartezeit, welche er in der `Waitq` verbringt, ist hier für das gesetzte Untersuchungsziel relevant.

```
class Client: public Process {
public:
    Client( Simulation* sim, Label l ): Process( sim, l ) {}
    virtual int main() {
        ::clientQ -> wait();
        return 0;
    }
};
```

Quelle 4.6: `Client`-Prozess

Neben diesen Prozesstypen gibt es noch einen `Generator`-Prozess, der entsprechend eines zufälligen Ankunftszeitintervalls neue Clients erzeugt.

```
class Generator: public Process {
public:
    Generator( Simulation* sim, Label l, Rdist* rand ):
        Process( sim, l ),  arrivalInterval ( rand ) {}
    virtual int main() {
        Process* client = 0;
        while( ::clientCount-- ) {
            holdFor(  arrivalInterval  -> sample() );
            client = new Client( getSimulation(), "Client" );
            client -> activate();
        }
        return 0;
    }
private:
    Rdist*  arrivalInterval ;
};
```

Quelle 4.7: `Generator`-Prozess

Das Simulationsprogramm verwendet in diesem Fall die von der Bibliothek bereitgestellte `DefaultSimulation`. Es werden alle beim Start nötigen Komponenten erzeugt und initialisiert: Zufallsvariablen, Reportobjekt, `Generator` und `Server`. Nach dem Simulationslauf wird ein Report erstellt, der komponentenbezogene Simulationsstatistiken enthält (siehe Abbildung 4.4).

4.6.2 Ereignisbasiertes Modell

Die Modellierung eines einfachen *Client-Server*-Systems entsprechend des oben genannten Untersuchungsziels der Wartedauer kommt mit zwei Ereignissen aus. Das ist zum einen das Eintreffen einer Client-Anfrage und zum anderen die fertige Bearbeitung einer Anfrage durch den Server. Die Klassen Server und Client sind in diesem Modell keine aktiven Komponenten, sondern lediglich Datenstrukturen, welche die Statistikerstellung erleichtern.

Das in Quelle 4.8 dargestellte Ankunftsereignis ist das zentrale Modellelement. Es wird ein neuer `Client` erzeugt, der entweder in die Warteschlange eingereiht oder dem Server zugeteilt wird. Im zweiten Fall erfolgt dann auch gleich die Einplanung des Bearbeitungsende-Ereignisses. Weiterhin trägt sich das Ankunftsereignis erneut selbst im Ereigniskalender ein, falls noch weitere Clients zu erzeugen sind.

```
class ClientArrival : public Event {
public:
    ClientArrival ( Simulation* sim, Label l, Rdist* arr, Rdist* serv, Server* s ):
        Event( sim, l ), arrivalInterval ( arr ), servicePeriod ( serv ), server ( s ) {
            finishEvent = new ClientFinished( sim, "ClientFinished", servicePeriod, server );
        }
    virtual void eventAction() {
        Client* newClient = new Client( getSimulation(), "Client", getCurrentTime() );
        if ( server -> idle ) {
                server -> worksOn( newClient );
                finishEvent -> scheduleIn( servicePeriod -> sample() );
        } else { ::queue -> push( newClient );
                :: reporter -> queueStats.update( ::queue -> size() );
        }
        if ( --::clientCount ) scheduleIn( arrivalInterval -> sample() );
    }
private:
    Rdist* arrivalInterval, *servicePeriod;
    Server* server;
    Event* finishEvent;
};
```

Quelle 4.8: `ClientArrival`-Ereignis

```
class ClientFinished : public Event {
public:
    ClientFinished ( Simulation* sim, Label l, Rdist* servTime, Server* s ):
        Event( sim, l ), servicePeriod ( servTime ), server ( s ) {}
    virtual void eventAction() {
        server -> finishedClient();
        if ( ::queue -> empty() ) server -> idle = true;
        else {
            if ( :: reporter -> maxQueueLength < ::queue -> size() )
                :: reporter -> maxQueueLength = ::queue -> size();
            server -> worksOn( ::queue -> front() ); ::queue -> pop();
            :: reporter -> queueStats.update( ::queue -> size() );
            this -> scheduleIn( servicePeriod -> sample() );
        }
    }
private:
    Rdist* servicePeriod;
    Server* server;
};
```

Quelle 4.9: `ClientFinished`-Ereignis

Das Bearbeitungsende wird durch ein `ClientFinished`-Ereignis signalisiert, dargestellt in Quelle 4.9. Die Bindung zwischen Server und bearbeitetem Client wird gelöst, der Zustand des Servers nach *frei* gewechselt oder ein neuer Client aus der Warteschlange zugeteilt, womit sich auch das Bearbeitungsende-Ereignis erneut selbst einplant.

Der Komfort, den die `Waitq` im Prozessmodell hinsichtlich automatisierter Statistik bietet, muss bei der ereignisbasierten Variante in diesem Fall von Hand hinzugefügt werden. Dies geschieht durch die in Quelle 4.10 gezeigte Klasse `Stat`, welche die Funktionalität eines Berichterzeugers implementiert.

```
class Stat: public ReportProducer {
public:
    unsigned int synchs, zeroClients, zeroServers, maxQueueLength;
    SimTime sumClientWait, sumServerWait;
    Accum queueStats;

    Stat(): synchs( 0 ), zeroClients( 0 ), zeroServers( 0 ),
            maxQueueLength( 0 ), sumClientWait( 0 ), sumServerWait( 0 ),
            queueStats( getDefaultSimulation(), "ClientQueueStats" ) {}

    double getAvgClientWaitTime() const { return sumClientWait / synchs; }
    double getAvgServerWaitTime() const { return sumServerWait / synchs; }
    double getAvgLength() const { return queueStats.getMean(); }

    void report( Report* r ) {
        assert(r);
        static const char* labels[] = {
            "Name", "Number of Synchs", "Zero wait servers", "Avgg server wait",
            "Zero wait clients", "Avgg client wait", "Max client queue length",
            "Now client queue length", "Avgg client queue length" };
        static const ColumnType types[] = { STRING, INTEGER, INTEGER, REAL,
                                            INTEGER, REAL, INTEGER, INTEGER, REAL };
        static utTableDef def( sizeof(labels)/sizeof(char*), labels, types );
        Table* t = r -> createTable( "Event Simulation Statistics", &def );
        *t << "Event Simulation Stat"<< synchs << zeroServers << getAvgServerWaitTime();
        *t << zeroClients << getAvgClientWaitTime() << maxQueueLength;
        *t << ::queue -> size() << getAvgLength();
    }
};
```

Quelle 4.10: `Stat`-Klasse für die Statistik

Die Datenstrukturen `Server` und `Client` dienen hier der Vereinfachung beim Sammeln statistischer Daten, damit die Lesbarkeit der `eventAction()`-Funktionen nicht zu sehr

beeinträchtigt wird. Dabei braucht ein `Client` nur seine Ankunftszeit zu vermerken.

```
class Client : public DefLabeledObject {
public:
    Client( Simulation* sim, Label l, SimTime t ):  arrivalTime( t )
        { DefLabeledObject::setLabel( sim, l ); }
    SimTime arrivalTime;
};
```

Quelle 4.11: `Client`-Klasse für die Statistik

Der `Server` dagegen meldet dem Berichterzeuger `Stat* reporter` Daten zu Wartezeiten und Synchronisation. Zusätzlich wird auch hier der Zustand des Servers in einem Memberdatum gespeichert und der aktuelle `Client` vermerkt.

```
class Server : public DefLabeledObject {
public:
    Server( Simulation* sim, Label l ):
        idle( true ), sim( sim ), finishTime( 0 ), currentClient ( 0 )
        { DefLabeledObject::setLabel( sim, l ); }
    void worksOn( Client* c ) {
        if ( idle ) {
            SimTime sWaitTime = sim -> getTime() - finishTime;
            :: reporter -> sumServerWait += sWaitTime;
            :: reporter -> zeroClients++;
        }
        else {
            :: reporter -> zeroServers++;
        }
        idle = false;
        currentClient = c;
        SimTime cWaitTime = sim -> getTime() - c -> arrivalTime;
        :: reporter -> sumClientWait += cWaitTime;
        :: reporter -> synchs++;
    }
    void finishedClient () { finishTime = sim -> getTime(); }
    bool idle;
    Simulation* sim;
    SimTime finishTime;
    Client* currentClient;
};
```

Quelle 4.12: `Server`-Klasse für die Statistik

Das Simulationsprogramm verwendet auch in diesem Modell die bereitgestellte `DefaultSimulation`. Die zum Start nötigen Komponenten sind ähnlich. Anstelle des

Client-Generators wird hier das erste Ankunftsereignis zu einem zufälligen Zeitpunkt eingeplant. Für die Statistik werden ein `Stat`-Objekt und der `Server` initialisiert. Als Vergleichsmöglichkeit beider Simulationsmodelle wird auch in diesem Simulationsprogramm am Ende ein Report erstellt (siehe Abbildung 4.5).

4.6.3 Erkenntnisse

Vergleicht man die Abbildungen 4.4 und 4.5, welche die Berichte beider Simulationen zeigen, so ist festzustellen, dass sie gleiche Resultate liefern. Das Untersuchungsziel war die Beantwortung der Frage nach der durchschnittlichen Wartezeit der einzelnen Clients, welche man nach Simulation der Bearbeitung von 2000 Client-Anfragen in der `Waitq`-Statistik ablesen kann. Die durchschnittliche *Slave*-Wartezeit betrug 20.8158 Zeiteinheiten.

Das gleiche Ergebnis von 20.8158 Zeiteinheiten Client-Wartezeit ist auch im Bericht des `Stat`-Objekts der Ereignissimulation zu finden. Daneben finden sich wie bei der `Waitq` auch weitere Daten zur Anzahl der Synchronisationen, zur Wartezeit des Servers und zur durchschnittlichen Länge der Client-Warteschlange.

Analysiert man allerdings die Laufzeiten beider Modelle, so unterscheiden diese sich mit zunehmender Client-Anzahl deutlich. Die Messungen wurden mit Version 2.0 der ODEMx-Bibliothek auf dem selben Rechner durchgeführt. Die Zeitausschriften wurden durch das Linux-Kommando `time` erzeugt, welches zum Messen der Laufzeit einfacher Anwendungen verwendet werden kann.

```
****************************** Prozess-Simulation ******************************

|2000 Clients                  |100000 Clients                |500000 Clients
|                              |                              |
| real    0m0.440s             | real    0m22.986s            | real    2m22.814s
| user    0m0.380s             | user    0m22.697s            | user    2m20.837s
| sys     0m0.008s             | sys     0m0.172s             | sys     0m0.776s


****************************** Ereignis-Simulation ******************************

|2000 Clients                  |100000 Clients                |500000 Clients
|                              |                              |
| real    0m0.271s             | real    0m3.956s             | real    0m20.308s
| user    0m0.192s             | user    0m3.884s             | user    0m19.849s
| sys     0m0.000s             | sys     0m0.016s             | sys     0m0.036s
```

Quelle 4.13: Laufzeitvergleich Prozessmodell - Ereignismodell

Simulation: DefaultSimulation **SimTime: 4040.95**

HtmlReport **ODEMx Version: 2.0**

Queue Statistics

Name	Reset at	Min queue length	Max queue length	Now queue length	Avg queue length
ClientQueue_master_queue	0	0	1	1	0.0161295
ClientQueue_slave_queue	0	0	29	0	10.3024

Waitq Statistics

Name	Reset at	Master Queue	Slave Queue	Number of Synch.	Zero wait masters	Avg masters wait	Zero wait slaves	Avg slaves wait
Client Queue	0	ClientQueue master_queue	ClientQueue slave_queue	2000	1919	0.0325893	81	20.8158

Random Number Generators

Name	Reset at	Type	Uses	Seed	Parameter 1	Parameter 2	Parameter 3
ArrivalInterval	0	Uniform	2000	33427485	1	3	0
ServicePeriod	0	Uniform	2000	22276755	0.5	3.5	0

Abbildung 4.4: HTML-Report zum prozessorientierten Beispiel

Simulation: DefaultSimulation **SimTime: 4040.95**

HtmlReport **ODEMx Version: 2.0**

Event Simulation Statistics

Name	Number of Synchs	Zero wait servers	Avg server wait	Zero wait clients	Avg client wait	Max client queue length	Now client queue length	Avg client queue length
Event Simulation Stat	2000	1919	0.0325893	81	20.8158	29	0	10.3024

Random Number Generators

Name	Reset at	Type	Uses	Seed	Parameter 1	Parameter 2	Parameter 3
ArrivalInterval	0	Uniform	2000	33427485	1	3	0
ServicePeriod	0	Uniform	2000	22276755	0.5	3.5	0

Abbildung 4.5: HTML-Report zum ereignisorientierten Beispiel

Selbst bei der geringen Anzahl 2000 simulierter Client-Anfragen ist der Unterschied schon deutlich zu erkennen. Das Prozessmodell benötigt fast die doppelte Laufzeit der Ereignisversion. Bei Skalierung des Beispiels auf 500000 Anfragen wird allerdings der Unterschied noch viel deutlicher mit der siebenfachen Laufzeit in der Prozessversion.

Anhand dieses einfachen Beispiels lässt sich feststellen, dass die Verwendung unterschiedlicher Modelle durchaus sinnvoll ist, denn häufig kommt es bei der Simulation auch darauf an, die Laufzeit zu minimieren, um die Durchführung ganzer Experiment-Serien in akzeptabler Laufzeit zu erreichen. Für das vorgestellte Client-Server-System wäre eventuell aus diesem Grund die Verwendung des ereignisbasierten Modells vorzuziehen.

Andererseits wird für die rein ereignisbasierte Modellierung meist auch mehr Aufwand betrieben als beim Prozessmodell, da die Zusammenhänge zwischen Ereignissen und Systemkomponenten nicht immer leicht festzustellen sind und einen Gesamtüberblick des Systems verlangen. Die Umsetzung des obigen Beispiels bleibt allein deshalb recht einfach, weil das Verhalten der einzelnen Komponenten `Client` und `Server` für das gewählte Untersuchungsziel nicht von Belang ist.

Ein weiterer zusätzlicher Aufwand entstand durch die Statistik, welche im Ereignismodell von Hand hinzugefügt wurde, da Komponenten mit automatisierter Statistik wie die Warteschlange `Waitq` aktuell in ODEMx nur für Prozesse ausgelegt sind.

5 Implementierung von Timern in ODEMx

5.1 Das Konzept

Zur Prozesssynchronisation ist es manchmal notwendig, dass ein Prozess auf das Eintreten bestimmter Bedingungen wartet, bevor er mit seinem Lebenszyklus fortfahren kann. Das kann unter anderem die Verfügbarkeit einer Ressource oder die Ankunft einer Nachricht sein, wie es bei der Kommunikation über begrenzte Puffer der Fall ist. Sollte allerdings ein Prozess auf eine nie eintretende Situation warten, so tritt eine permanente Blockierung ein.

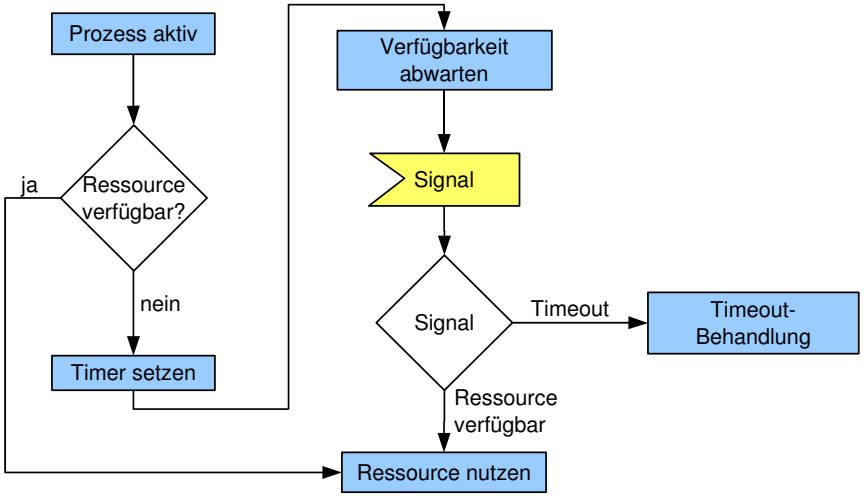

Abbildung 5.1: Allgemeine Darstellung des Timer-Konzepts

Abbildung 5.1 zeigt schematisch die Abfolge des Timeout-überwachten Wartekonzepts. Wenn die verlangte Ressource nicht verfügbar ist, wird ein Timer gesetzt, der die Wartedauer beschränkt. Die Wartephase wird in jedem Fall durch ein Signal beendet, welches

entweder anzeigt, dass die Ressource verfügbar geworden ist oder dass ein Weckereignis ausgelöst wurde. So kann eine Prozessblockierung aufgehoben werden, womit sichergestellt ist, dass ein Prozess spätestens nach einem Timeout seinen Lebenszyklus fortsetzen kann.

5.2 Timer in ODEM

Wie in Abschnitt 4.1 beschrieben, verwaltet der Ereigniskalender der Bibliothek ODEM über polymorphe Zeiger Objekte der Typen `Process` und `Timer`. Alle Objekte dieser Typen werden bezüglich der Scheduling-Operationen gleichartig behandelt. Allerdings haben Timer eine höhere Priorität, damit sie zu ihrem Auslösungszeitpunkt vor Prozessen mit derselben Ausführungszeit ausgelöst werden.

Grundlage für die Implementierung des Konzepts ist die Klasse `Memo`, mit Hilfe derer sich beliebig viele Prozessobjekte speichern lassen, die dann bei Eintreten eines vordefinierten Ereignisses per Funktionsruf aufgeweckt werden können. Sowohl Prozesse als auch Timer sind in ODEM davon abgeleitet. Dadurch können `Timer`-Objekte in ODEM eine Liste von Prozessen speichern, die bei Timeout zu alarmieren sind. Alarmierung bedeutet in dem Fall den Neueintrag der vermerkten Prozesse im Ereigniskalender zum aktuellen Simulationszeitpunkt, die Reihenfolge (LIFO oder FIFO) wird dabei durch einen internen Schalter bestimmt.

Die `Memo`-Funktionalität ist ein wichtiger Bestandteil des gesamten oben beschriebenen Wartekonzepts, weil die Klasse `Process` für den Start der Wartephase eine Memberfunktion `wait()` bereitstellt, die beliebig viele `Memo`-Objekte als Parameter haben kann, welche den Prozess aufwecken können. Wird ein Prozess auf diese Weise suspendiert, so erfolgt seine Reaktivierung, sobald eines der übergebenen Objekte verfügbar wird und den Prozess alarmiert. Detaillierte Beschreibungen der Klassen `Timer` und `Memo` in ODEM sind in [FA96] zu finden.

5.3 Ansatz für Timer in ODEMx

Da es sich bei Timern um Zeitereignisse handelt, werden sie in ODEMx als Ableitung von `Event` eingeführt. Eine weitere Basisklasse für `Timer` ist die Klasse `Memo`, welche ebenfalls in ODEMx reimplementiert wird. Die Hierarchie der Klassen in ODEMx ist in Abbildung 5.2 dargestellt.

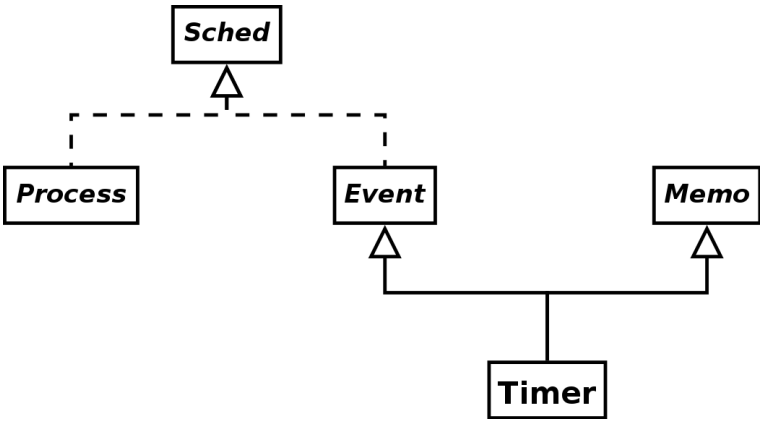

Abbildung 5.2: Ausschnitt der erweiterten Klassenhierarchie von ODEMx

An der Klasse `Process` werden einige Erweiterungen vorgenommen, um das Konzept aus 5.1 unter Verwendung von `Memo` zu implementieren. Da dieses Konzept zur Synchronisation von Prozessen und Ressourcen dient, werden alle in diesem Kapitel beschriebenen Modifikationen an ODEMx als eine Erweiterung des Moduls Synchronization implementiert (siehe Abschnitt 3.2.2).

5.4 Neue und veränderte Klassen

5.4.1 Die Klasse `Memo`

```
class Memo: public Observable<MemoObserver>, public virtual TraceProducer {
protected:
    friend class Process;

    Memo( Simulation* sim, MemoType t, MemoObserver* mo = 0 );
    virtual ~Memo();
    bool remember( Sched* newObject );
    bool forget( Sched* rememberedObject );
    void eraseMemory();
    virtual bool available() = 0;
    virtual void alert();
    ...
    std::list<Sched*> memoryList;
};
```

Quelle 5.1: Klassendeklaration `Memo`

Die Funktionalität eines Prozessgedächtnisses von Objekten wird analog zu ODEM durch die abstrakte Klasse `Memo` implementiert, welche die Synchronisation von Prozessen und passiven Strukturen wie gepufferten Kommunikationsports realisiert.[1]

Es können beliebig viele Prozessobjekte in der internen Liste `memoryList` unter Verwendung der Funktionen `remember()`, `forget()` und `eraseMemory()` verwaltet werden. Ist eine begrenzte Ressource als Ableitung von `Memo` implementiert, so wird durch die rein virtuelle Funktion `available()` ein Test auf Verfügbarkeit bereitgestellt, welcher in `Process::wait()` vor der Suspendierung eines Prozesses abgefragt wird – `available()` muss daher in jeder abgeleiteten Klasse implementiert sein (siehe Abschnitt 5.4.3).

Sobald im Simulationsverlauf bestimmte vordefinierte Situationen eintreten, wo suspendierte vermerkte Prozesse ihren Lebenszyklus fortsetzen sollen, können durch Ruf von `alert()` alle vermerkten Prozesse zu genau diesem Simulationszeitpunkt wieder im Ereigniskalender eingeplant werden, wonach sie aus dem `Memo`-Objekt gelöscht werden. Die Funktion ist virtuell und damit polymorph verwendbar, womit alle `Memo`-Ableitungen auch eine angepasste Version von `alert()` definieren können.

5.4.2 Die Klasse `Timer`

```
class Timer: public Event, public Memo, public Observable<TimerObserver> {
public:
    Timer( Simulation* sim, Label l, TimerObserver* to = 0 );
    virtual ~Timer();
    void setIn( SimTime t );
    void setAt( SimTime t );
    void reset( SimTime t );
    void stop();
    bool isSet();
    bool registerProcess( Process* p );
    bool removeProcess( Process* p );
    virtual void eventAction();
    virtual bool available();
    virtual Trace* getTrace() const;
    virtual Label getLabel() const;
};
```

Quelle 5.2: Klassendeklaration `Timer`

[1] Kommunikationsports basieren auf der Implementierung beschränkter Puffer von Hansen (siehe [Ha90])

Die Klasse `Timer` ist als Ableitung von `Event` implementiert und erbt so die Scheduling-Funktionalität. Bereits mit dem Konstruktor wird die Priorität jedoch auf den Maximalwert gesetzt, damit sichergestellt ist, dass Timer zu ihrem Ausführungszeitpunkt als erstes Ereignis ausgelöst werden.

Das Setzen eines Timers erfolgt durch die Zugriffsoperationen `setIn(t)` oder `setAt(t)`, für Timeout zum Simulationszeitpunkt `now + t` bzw. dem absoluten Zeitpunkt `t`. Zur Ausdehnung der Wartezeit von Timern wird die Funktion `reset(t)` bereitgestellt, die auf bereits gestartete Timer angewendet werden kann. Ob ein Timer aktuell läuft, kann durch `isSet()` ermittelt werden. Gestoppt werden Timer mit `stop()`, wobei dann auch alle vermerkten Prozesse automatisch aus dem Memberdatum `Memo::memoryList` entfernt werden. Die zum Timeout-Ereignis gehörige Aktion wird in der Funktion `eventAction()` beschrieben, da deren Reimplementierung von der abstrakten Basisklasse `Event` vorgeschrieben ist.

Zur Vereinfachung einer anderweitigen Verwendung von `Timer` werden noch die Funktionen `registerProcess()` und `removeProcess()` bereitgestellt. Allerdings ist es normalerweise bei Verwendung des bereitgestellten Wartekonzepts nicht notwendig, Prozesse explizit in einem Timer zu vermerken, wie der nächste Abschnitt zeigt.

5.4.3 Die Klasse `Process`

```
class Process : public Sched, public Coroutine, public Observable<ProcessObserver> {
public:
    Memo* wait( Memo* m0, Memo* m1 = 0, Memo* m2 = 0,
                Memo* m3 = 0, Memo* m4 = 0, Memo* m5 = 0 );
    Memo* wait( MemoVector* memvec );
    virtual void alertProcess ( Memo* theAlerter );
    bool isAlerted () const;
    Memo* getAlerter() const;
    void resetAlert ();
    ...
private:
    bool alerted ;
    Memo* alerter;
    bool internalMemVec;
    ...
};
```

Quelle 5.3: Änderungen an `Process`

Das in Abschnitt 5.1 vorgestellte Timeout-überwachte Wartekonzept verlangt einige Erweiterungen an der Klasse `Process`. Analog zu ODEM wird die Memberfunktion `Process::wait()` eingeführt, welche auf der Basis von `Memo` die gewünschte Funktionalität erbringt. Es gibt zwei Versionen der Funktion, wobei der ersten Version bis zu sechs einzelne `Memo`-Objekte übergeben werden können. Intern wird dann ein `MemoVector` daraus konstruiert und die zweite Version damit aufgerufen.

Die in diesem Abschnitt erörterte Darstellung der Funktion `wait()` zeigt die Grundfunktionalität ohne Speicher-Management, *Trace*-Markierungen und *Observation*-Ereignisse, welche an dieser Stelle irrelevant sind.

Beim Aufruf von `wait()` wird ein Vektor mit `Memo`-Zeigern übergeben, dessen Elemente zunächst auf Verfügbarkeit getestet werden. Ist dies der Fall, so wird die Funktion umgehend mit Rückgabe des ersten verfügbaren `Memo`-Objekts beendet, weil eine von dem Prozess verlangte Ressource nutzbar ist. Bei Timern ist Verfügbarkeit nur dann gegeben, wenn sie nicht gesetzt sind, weshalb an dieser Stelle eine Warnung erfolgt.

```
Memo* Process::wait( MemoVector* memvec ) {
    MemoVector::iterator  iter ;
    for ( iter = memvec -> begin(); iter != memvec -> end(); ++iter ) {
        if ( (* iter ) -> available() ) {
            if ( (* iter ) -> getMemoType() == Memo::TIMER )
                warning( "Process::wait(): Timer is available due to not being set" );
            return (* iter );
        }
    }
...
```

Ist zu diesem Zeitpunkt keins der angegebenen `Memo`-Objekte verfügbar, so wird der Prozess automatisch in allen übergebenen Objekten mittels `Memo::remember()` vermerkt und nach Rücksetzung der Alarmdaten durch Ruf von `Process::sleep()` suspendiert.

```
...
    for ( iter = memvec -> begin(); iter != memvec -> end(); ++iter ) {
        (* iter ) -> remember( this );
    }
    resetAlert ();
    sleep ();
...
```

Wird die Funktion nach dieser Stelle fortgesetzt, so wurde der Prozess gerade reaktiviert. Es sind in ODEMx zwei Mechanismen vorgesehen, wodurch die Wartephase von Prozessen unterbrochen werden kann: Senden eines Alarms oder Senden eines Interrupts. Im ersten

Fall ist der Sender des Alarms ein `Memo`-Objekt und im zweiten Fall ist der Unterbrecher ein anderes `Sched`-Objekt. Liefert `isAlerted() true`, so wurde der Prozess mit Alarm geweckt, woraufhin er aus allen anderen `Memo`-Objekten gelöscht und der Alarmierer zurückgegeben wird.

```
...
    if ( isAlerted () ){
        Memo* currentAlerter = getAlerter();
        for ( iter = memvec -> begin(); iter != memvec -> end(); ++iter ) {
            if ((* iter ) != currentAlerter )
                (* iter ) -> forget(this);
        }
        resetAlert ();
        return currentAlerter ;
    }
...
```

Sonst liefert `wait()` immer den Rückgabewert 0, woraufhin jeder Prozess nach Rückkehr aus dieser Funktion dann auf Interrupt testen sollte. Es erfolgt jedoch auch eine bibliotheksseitige Warnung.

```
...
    else if ( isInterrupted () ) {
        for ( iter = memvec->begin(); iter != memvec->end(); ++iter )
            (* iter ) -> forget(this);
        warning( "Process::wait(): waiting period ended by interrupt" );
        return 0;
    }
    else
        error( "Process::wait(): Process awakened without alert or interrupt" );
    return 0;
}
```

Das Wecken durch eine andere Scheduling-Funktion ist nicht vorgesehen und führt zu einer Fehlerausschrift.

Zur Alarmierung stellt die Klasse `Process` die Memberfunktion `alertProcess()` bereit, welche in `Memo::alert()` verwendet wird. Das aufrufende `Memo`-Objekt wird dabei als Parameter weitergereicht. Im `Process`-Objekt wird damit vermerkt, dass es durch einen Alarm aktiviert wurde und wer der Alarmierer ist. Anschließend wird der Prozess zum aktuellen Simulationszeitpunkt im Ereigniskalender eingetragen.

```
void Process :: alertProcess ( Memo* theAlerter ) {
    if ( !setProcessState(RUNNABLE) ) return;
    alerted = true;
    alerter = theAlerter;
    setExecutionTime( getCurrentTime() );
    env -> insertSched( this );
}
```

Sobald der Prozess dann von der Simulation wieder aktiviert wird, kann er durch `isAlerted()` und `getAlerter()` diese privaten Memberdaten abfragen und mit einer Fallunterscheidung sein Verhalten dem Resultat entsprechend fortsetzen.

5.5 Unterstützung der Auswertungsmöglichkeiten

Die für das Wartekonzept hinzugefügten Klassen und auch die Erweiterungen an `Process` unterstützen allesamt sowohl *Trace* als auch *Observation*. Wie für die in Abschnitt 4.5 beschriebenen Klassen ist der Quellcode dazu im Anhang A.1 zu finden.

Memo Als neue Komponente sendet diese Klasse *Trace*-Markierungen und auch *Observation*-Ereignisse. Sie ist abgeleitet von `TraceProducer` und `Observable<>`. Die Schnittstelle für `MemoObserver` meldet die Aufrufe von Konstruktor und Destruktor sowie von `remember()`, `forget()` und `alert()`. Gleichzeitig senden diese Funktionen auch *Trace*-Markierungen zur Protokollierung dieser Simulationsereignisse.

Timer Auch Timer fungieren als `TraceProducer` und `Observable<>`. Die Schnittstelle `TimerObserver` erbt von den Schnittstellen `EventObserver` und `MemoObserver`, damit durch einen einzelnen Beobachter auch *Observation*-Ereignisse der Basisklassen verarbeitet werden können. Beobachtbar sind an `Timer`-Objekten die Funktionen zum Setzen und Stoppen eines Timers, die Prozessspeicherung und -löschung sowie die Auslösung des Timeout-Ereignisses.

Process Als Basisklasse für Prozesse unterstützte `Process` bereits vor der Bibliothekserweiterung ausgiebig die Konzepte *Trace* und *Observation*, so dass nur einige Ergänzungen gemacht wurden, die das neu eingeführte Timeout-überwachte Wartekonzept betreffen. So kann ein `ProcessObserver` nun den Aufruf von `wait()` und `alertProcess()` beobachten und es werden *Trace*-Markierungen bei Warten, Alarmierung und Verfügbarkeit

eines `Memo`-Objekts versandt.

5.6 Anwendungsbeispiel

Das Beispiel beschreibt einen Prozess `Receiver`, der einen Empfangsport für Nachrichten hat. Erhält diese Komponente keine Nachricht innerhalb von 15 Zeiteinheiten, so wird ein Timeout ausgelöst. Um die Verhaltensbeschreibung einfach zu halten, wird der Prozess in diesem Fall sofort beendet.

```
class Receiver: public Process {
public:
    Receiver(): Process( getDefaultSimulation(), "Receiver") {
        myTimer = new Timer( getDefaultSimulation(), "Message Timer" );
        myPort = new PortHead( getDefaultSimulation(), "Input Port" );
    }
    virtual int main() {
        myTimer -> setIn( 15 );
        while ( true ) {
            Memo* who = wait( myTimer, myPort );
            if ( who == myTimer ) break;
            else if ( who == myPort ) {
                myTimer -> reset( 15 );
                Msg* m = (Msg*)myPort -> get();
            }
        }
        return 0;
    }
    PortTail* getInputPort() { return myPort -> getTail(); }
private:
    Timer* myTimer;
    PortHead* myPort;
};
```

Quelle 5.4: `Receiver`-Prozess

Zur Modellierung des Eintreffens einer Nachricht wird das Ereignis `MessageArrival` definiert, welches die Ankunft einer Nachricht `m` bei einem zugeordneten `Receiver p` beschreibt.

```
class MessageArrival: public Event {
public:
    MessageArrival( Receiver* p, Msg* m ):
        Event( getDefaultSimulation(), "Message Arrival" ),
        myReceiver( p ), myMessage( m ) {}
    virtual void eventAction() {
        myReceiver -> getInputPort() -> put( myMessage );
    }
private:
    Receiver* myReceiver;
    Msg* myMessage;
};
```

Quelle 5.5: `MessageArrival`-Ereignis

Nachrichten sind hier durch eine einfache C++-Struktur modelliert, wobei `PortElement` die Basisklasse für `Port`-Inhalte ist.

```
struct Msg: public PortElement {
    Msg( char* str ): content( str ) {}
    char* content;
};
```

Quelle 5.6: `Msg`-C++-Struktur

Anhand des `HtmlTrace`, gezeigt in Abbildung 5.3, lässt sich folgender Simulationsverlauf nachvollziehen: Zum Zeitpunkt 0 wird der *Receiver* aktiviert und die Nachrichtenankunft zum Zeitpunkt 8 festgelegt. Der *Receiver* setzt seinen *Message Timer* und wartet durch Ruf von `wait()`. Dabei wird er automatisch in den `Memo`-Objekten *Message Timer* und *Input Port* vermerkt und anschließend suspendiert. Bei Nachrichtenankunft sendet der *Input Port* einen Alarm, durch den der *Receiver* wieder aktiviert wird. Bei Fortsetzung seines Lebenszyklus' kann er dann die Nachricht verarbeiten und anschließend seinen *Message Timer* durch `reset()` für den nächsten potenziellen Timeout vorbereiten. Danach wartet der *Receiver* auf eine weitere Nachricht. Da diese bis zum Timeout-Zeitpunkt nicht empfangen wird, weckt der *Message Timer* den Prozess auf, woraufhin dieser seinen Lebenszyklus beendet.

Time	Sender	Event	Details		Comment
0	Receiver	activate	Partner	DefaultSimulation	
	Message Arrival	scheduleAt	current	DefaultSimulation	
			absolute Time	8	
	Receiver	execute	Partner	DefaultSimulation	
	Message Timer	set			
		scheduleIn	current	Receiver	
			relative Time	15	
	Receiver	wait	partner	Message Timer	
			partner	Input Port	
	Message Timer	remember	Partner	Receiver	
	Input Port	remember	Partner	Receiver	
	Receiver	sleep	Partner	Receiver	
8	Message Arrival	execute	Partner	Receiver	
	Input Port	alert	partner	Receiver	
	Receiver	alert			
		execute	Partner	Message Arrival	
	Message Timer	forget	Partner	Receiver	
		reset			
		scheduleIn	current	Receiver	
			relative Time	15	
	Receiver	wait	partner	Message Timer	
			partner	Input Port	
	Message Timer	remember	Partner	Receiver	
	Input Port	remember	Partner	Receiver	
	Receiver	sleep	Partner	Receiver	
23	Message Timer	execute	Partner	Receiver	
		timeout			
		alert	partner	Receiver	
	Receiver	alert			
		execute	Partner	Message Timer	
	Input Port	forget	Partner	Receiver	
Time	Sender	Event	Details		Comment

Abbildung 5.3: `HtmlTrace`-Ausgabe für Receiver-Beispiel

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ODEMx, eine in C++ geschriebene Bibliothek zur computergesteuerten Simulation, um ein Modellierungsparadigma erweitert. Ließen sich bisher ausschließlich prozessbasierte Systemmodelle mit ODEMx modellieren und simulieren, so gibt es mit der Erweiterung zu Version 2.0 nun auch die Möglichkeit, ereignisbasierte Systemmodelle zu verwenden. Darüber hinaus wird auch die Kombination beider Methoden in einem Modell unterstützt, was dem Modellierer die Wahl der passenden Bibliothekskomponenten erlaubt, ohne ihn auf eine bestimmte Modellierungsart einzuschränken.

Aufbauend auf Konzepten der Vorgängerbibliothek ODEM wurde zur Implementierung von Ereignissen die verwendete Klassenhierarchie um eine Interface-Klasse erweitert, deren Ableitungen nun die Vorlagen für aktive Objekte in Simulationsmodellen darstellen. Weiterhin wurde der Scheduling-Apparat von ODEMx analysiert und für die Verwendung von Ereignissen erweitert.

Beim Übergang von ODEM zu ODEMx im Jahre 2003 fanden nicht alle Konzepte ihren Weg in die neue Bibliothek [Ge03]. Daher wurde im Rahmen dieser Arbeit beispielhaft auf Grundlage der neu eingeführten allgemeinen Ereignisklasse ein allgemeines Timeoutüberwachtes Wartekonzept aus ODEM in ODEMx reimplementiert.

Da die Analyse von Simulationen erheblich von der Protokollierung von Zustandsänderungen während des Simulationslaufs abhängt, wurde bei allen neuen Bibliothekselementen und bei Erweiterungen vorhandener Klassen insbesondere Wert darauf gelegt, dass die Auswertungsmechanismen der Bibliothek ODEMx durchgängig unterstützt werden.

Hinsichtlich der Laufzeit bleibt festzustellen, dass die konsequente Nutzung der Auswertungsmechanismen, die das Simulationsgeschehen detailliert erfassen, zu Laufzeiteinbußen durch eine höhere Anzahl von Funktionsrufen führt. Davon betroffen sind insbesondere *Trace* und *Observation*, weil jede erfasste Zustandsänderung im Modell mindestens einen Funktionsaufruf für die *Trace*-Markierung und einen weiteren für Beobachter-Objekte er-

fordert.

Positiv zu vermerken ist jedoch, dass geschickte Anwendung unterschiedlicher Modellierungsparadigmen auch zu deutlichen Laufzeitverbesserungen in der Simulation mit ODEMx führen kann, wie das vergleichende Anwendungsbeispiel in Abschnitt 4.6 verdeutlicht.

6.2 Ausblick: Weitere Verbesserungen

Bei der Arbeit mit der Bibliothek und der Analyse von Simulationen ergeben sich verschiedene Verbesserungsansätze. Da die Sprache C++ keine automatische Speicherbereinigung (*garbage collection*) bereitstellt, die während des Programmlaufs den Speicher aufräumt, muss das Speichermanagement von Hand betrieben werden. Der Grundsatz der möglichst einfachen Modellierung wird so etwas behindert, weil sich der Verfasser eines Simulationsprogramms nicht ausschließlich auf sein Modell konzentrieren kann, sondern eben auch auf Besonderheiten der Sprache C++ achten muss. Ein einfaches bibliotheksseitiges Speichermanagement, das alle zu einer Simulation gehörigen Objekte verwaltet, könnte diese Arbeit erleichtern.

In der Laufzeit verschiedener Simulationsmodelle treten beobachtbare Unterschiede auf und nicht alle Modelle skalieren gut – werden zum Beispiel sehr viele Prozesse in einem Modell verwendet, so zeigen Profiling-Tools wie der GNU Profiler *gprof*, wo Verbesserungsansätze hinsichtlich intern verwendeter Strukturen gegeben sind. Da möglichst kurze Laufzeiten in der Simulation von Bedeutung sind und gerade dies einer der Gründe für die Verwendung von C++ als Sprache ist, wären Verbesserungen in diesem Bereich wünschenswert.

Weiterhin hat die Erstellung von Beispielsimulationen gezeigt, dass sich viele Schritte vom Modell bis zum ersten lauffähigen Simulationsprogramm wiederholen, weil immer ähnliche Strukturen verwendet werden, man sie jedoch ständig neu in C++ notieren muss. Abhilfe könnte durch einen Simulationsassistenten geschaffen werden: ein Programm, welches Angaben bezüglich der verwendeten Modellkomponenten wie Prozesse und Ereignisse in ODEMx-Quellcode-Schablonen umsetzt, so dass der Modellierer lediglich die Verhaltensdefinition implementieren muss. Durch derartige Hilfsmittel könnte auch die Zugänglichkeit der Bibliothek verbessert werden, da die Lernkurve für neue Nutzer nicht unerheblich ist.

Literaturverzeichnis

[Bi79] Birtwistle, Graham M.: *Discrete Event Modelling on Simula.* Macmillan Publishing Company, 1979

[FA96] Fischer, Joachim und Ahrens, Klaus: *Objektorientierte Prozeßsimulation in C++.* Addison-Wesley, 1996

[LK00] Law, Averill M. und Kelton, W.David: *Simulation Modeling and Analysis, Third Edition.* McGraw-Hill, 2000

[St97] Stroustrup, Bjarne: *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997

[Ha90] Hansen, Tony L.: *The C++ Answer Book.* Addison-Wesley, 1990

[vL92] von Löwis, Martin: *Porting the Coroutine Process Library.* Humboldt- Universität zu Berlin, 1992

[Ze00] Zeigler, Bernard P., Praehofer, Herbert und Kim, Tag Gon: *Theory of Modeling and Simulation, Second Edition.* Academic Press, 2000

[Ge03] Gerstenberger, Ralf: *ODEMx Neue Lösungen für die Realisierung von C++-Bibliotheken zur Prozesssimulation.* Humboldt-Universität zu Berlin, 2003

[Ev06] Eveslage, Ingmar: *Reimplementation einer Stahlwerkssimulation auf Basis der Simulationsbibliothek ODEMx.* Humboldt-Universität zu Berlin, 2006

[Sk97] Sklenar, Jaroslav: *Introduction to OOP in SIMULA.* 1997, http://staff.um.edu.mt/jskl1/talk.html (Abgerufen: 8. Juni 2007)

[Doxy] Dimitri van Heesch, http://www.doxygen.org (Abgerufen: 10. Juni 2007)

A Anhang

A.1 Observer-Schnittstellen und Trace-Markierungen

```
class SimulationObserver: public ExecutionListObserver,
                          public CoroutineContextObserver {
public:
    ...
    virtual void onExecuteEvent(Simulation* sender, Event* e) {}
};
```

Quelle A.1: `Simulation`: Beobachter-Erweiterung

```
...
const MarkType Simulation::markExecEvent = MarkType("execute event", baseMarkId+7, typeid(Simulation));
```

Quelle A.2: `Simulation`: Neue Trace-Markierung bei Auslösung eines `Event`

```
class ExecutionListObserver {
public:
    ...
    virtual void onAddSched(ExecutionList* sender, Sched* newSched) {}
    virtual void onInsertSched(ExecutionList* sender, Sched* newSched) {}
    virtual void onInsertSchedAfter(ExecutionList* sender, Sched* newSched, Sched* previousSched) {}
    virtual void onInsertSchedBefore(ExecutionList* sender, Sched* newSched, Sched* nextSched) {}
    virtual void onRemoveSched(ExecutionList* sender, Sched* removedSched) {}
};
```

Quelle A.3: `ExecutionList`: Veränderte Beobachter-Schnittstelle

```
class EventObserver {
public:
    virtual ~EventObserver() {}
    virtual void onCreate(Event* sender) {}
    virtual void onDestroy(Event* sender) {}
    virtual void onSchedule(Event* sender) {}
    virtual void onScheduleAt(Event* sender, SimTime t) {}
    virtual void onScheduleIn(Event* sender, SimTime t) {}
    virtual void onScheduleAppend(Event* sender) {}
    virtual void onScheduleAppendAt(Event* sender, SimTime t) {}
    virtual void onScheduleAppendIn(Event* sender, SimTime t) {}
    virtual void onRemoveFromSchedule(Event* sender) {}
    virtual void onExecuteEvent(Event* sender) {}
    virtual void onChangePriority(Event* sender, Priority oldPriority,
                                  Priority newPriority) {}
    virtual void onChangeExecutionTime(Event* sender, SimTime oldExecutionTime,
                                       SimTime newExecutionTime) {}
};
```

Quelle A.4: `Event`: Neue Beobachter-Schnittstelle

```
const MarkTypeId Event::baseMarkId = 2000;
const MarkType Event::markCreate("create", baseMarkId+1, typeid(Event));
const MarkType Event::markDestroy("destroy", baseMarkId+2, typeid(Event));
const MarkType Event::markSchedule("schedule", baseMarkId+3, typeid(Event));
const MarkType Event::markScheduleAt("scheduleAt", baseMarkId+4, typeid(Event));
const MarkType Event::markScheduleIn("scheduleIn", baseMarkId+5, typeid(Event));
const MarkType Event::markScheduleAppend("scheduleAppend", baseMarkId+6, typeid(Event));
const MarkType Event::markScheduleAppendAt("scheduleAppendAt", baseMarkId+7, typeid(Event));
const MarkType Event::markScheduleAppendIn("scheduleAppendIn", baseMarkId+8, typeid(Event));
const MarkType Event::markRemoveFromSchedule("removeFromSchedule", baseMarkId+11, typeid(Event));
const MarkType Event::markExecute("execute", baseMarkId+12, typeid(Event));
const MarkType Event::markChangePriority("changePriority", baseMarkId+13, typeid(Event));
const MarkType Event::markChangeExecutionTime("changeExecutionTime", baseMarkId+14, typeid(Event));

const TagId Event::baseTagId = 2000;
const Tag Event::tagCurrent( "current" );
const Tag Event::tagAbsSimTime( "absolute Time" );
const Tag Event::tagRelSimTime( "relative Time" );
const Tag Event::tagPartner( "partner" );
```

Quelle A.5: `Event`: Neue Trace-Markierungen relevanter Funktionsrufe

```
class MemoObserver {
public:
    virtual ~MemoObserver() {}
    virtual void onCreate(Memo* sender) {}
    virtual void onDestroy(Memo* sender) {}
    virtual void onRemember(Memo* sender, Sched* s) {}
    virtual void onForget(Memo* sender, Sched* s) {}
    virtual void onAlert(Memo* sender) {}
};
```

Quelle A.6: `Memo`: Neue Beobachter-Schnittstelle

```
const MarkTypeId Memo::baseMarkId=3000;
const MarkType Memo::markRemember( "remember", baseMarkId+1, typeid(Memo) );
const MarkType Memo::markForget( "forget", baseMarkId+2, typeid(Memo) );
const MarkType Memo::markAlert( "alert", baseMarkId+10, typeid(Memo) );

const TagId Memo::baseTagId = 3000;
const Tag Memo::tagPartner( "partner" );
```

Quelle A.7: `Memo`: Neue Trace-Markierungen relevanter Funktionsaufrufe

```
class TimerObserver: public EventObserver, public MemoObserver {
public:
    virtual void onCreate(Timer* sender) {}
    virtual void onDestroy(Timer* sender) {}
    virtual void onSet(Timer* sender, SimTime t) {}
    virtual void onReset(Timer* sender, SimTime oldTime, SimTime newTime) {}
    virtual void onStop(Timer* sender) {}
    virtual void onRegisterProcess(Timer* sender, Process* p) {}
    virtual void onRemoveProcess(Timer* sender, Process* p) {}
    virtual void onTimeout(Timer* sender) {}
};
```

Quelle A.8: `Timer`: Neue Beobachter-Schnittstelle

```
const MarkTypeId Timer::baseMarkId = 2500;
const MarkType Timer::markSet( "set", baseMarkId+1, typeid(Timer) );
const MarkType Timer::markReset( "reset", baseMarkId+2, typeid(Timer) );
const MarkType Timer::markStop( "stop", baseMarkId+3, typeid(Timer) );
const MarkType Timer::markTimeout( "timeout", baseMarkId+5, typeid(Timer) );
```

Quelle A.9: `Timer`: Neue Trace-Markierungen relevanter Funktionsaufrufe

```
class ProcessObserver : public CoroutineObserver {
public:
    ...
    virtual void onWait(Process* sender, MemoVector* memvec) {};
    virtual void onAlert(Process* sender, Memo* alerter) {};
};
```

Quelle A.10: `Process`: Beobachter-Erweiterung

```
...
const MarkType Process::markWait("wait", baseMarkId+22, typeid(Process));
const MarkType Process::markAvailable("found available", baseMarkId+23, typeid(Process));
const MarkType Process::markAlert("alert", baseMarkId+24, typeid(Process));
```

Quelle A.11: `Process`: Neue Trace-Markierungen des Wartekonzepts